Suivi de flux d'information correct pour les systèmes d'exploitation Linux

Laurent Georget

ANNÉE 2017

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale 601 « Mathématiques et Sciences et Technologies de l'Information et de la Communication (MATHSTIC) »

présentée par

Laurent GEORGET

préparée à l'unité de recherche IRISA (UMR 6074) au sein de l'équipe-projet commune CIDRE — Inria / CNRS / Université de Rennes 1 / CentraleSupélec

---

Suivi de flux d'information correct pour les systèmes d'exploitation Linux

Thèse soutenue à Rennes le 28 septembre 2017

devant le jury composé de :

Jean GOUBAULT-LARRECQ
Professeur des universités, ENS Paris-Saclay / rapporteur

Marie-Laure POTET
Professeure des universités, Ensimag – Grenoble INP / rapporteur

Erven ROHOU
Directeur de recherche, Inria / examinateur

Sarah ZENNOU
Ingénieure de recherche, Airbus / examinatrice

Mathieu JAUME
Maître de conférences, UPMC / directeur de thèse

Valérie VIET TRIEM TONG
Professeure associée, CentraleSupélec / directrice de thèse

Remerciements

De nombreuses personnes et institutions ont contribué à l'aboutissement des trois années de production scientifique matérialisées par cette thèse et les publications en page 181 ; cette page leur est dédiée.

En tout premier lieu, je souhaite remercier les membres de mon jury qui ont accepté d'évaluer ce travail : Mr Erven Rohou, qui a présidé le jury, et qui avait auparavant présidé le jury de ma soutenance de mi-parcours ; Mr Jean Goubault-Larrecq, qui a rapporté cette thèse, et dont les nombreuses questions, en me forçant à éclaircir ma pensée, m'ont aidé à mieux présenter ce travail ; Mme Marie-Laure Potet, également rapportrice de cette thèse ; Mme Sarah Zennou.

Je remercie mes directeurs et encadrants de thèse, qui ont tous contribué à la fois scientifiquement et humainement à ce doctorat. Mathieu dans le rôle du sémanticien, a contribué énormément à la formalisation et aux preuves des contributions de cette thèse. Je le remercie aussi pour les sessions de travail à Jussieu et les bô buns à côté des arènes de Lutèce, et pour son enthousiasme immodéré pour les styles de flèches LaTeX. Valérie qui a dirigé ma thèse localement a fourni à cette thèse la vision d'ensemble et l'histoire du projet Blare et de son modèle. Je souhaite la remercier pour le temps qu'elle a su me consacrer aux bons moments et la liberté d'action que mon caractère indépendant, ou sauvage, nécessite. Nos expérimentations culinaires et para-culinaires ont fourni les moments de relâchement nécessaires à ce que ce travail arrive à terme. Frédéric a encadré cette thèse en fournissant son expertise sur le noyau Linux et en suggérant de quoi rendre plus intelligible les formalisations proposées par Mathieu et moi. Je le remercie pour s'être toujours montré intéressé et enthousiaste vis-à-vis des idées et techniques d'implémentation que j'ai proposées. Je garderai le souvenir des moments passés à corriger des bugs ou à administrer des serveurs capricieux. Enfin, Guillaume a fourni à chaque étape cruciale de cette thèse son attention avisée, ses corrections et ses commentaires sur ma rédaction. Je lui dois beaucoup en ce qui concerne la qualité de mes publications. Je le remercie aussi pour ses éclaircissements et sa grande culture historique et du droit dont mes pauses café ont bénéficiés.

Je remercie du fonc du cœur toute ma famille et mes amis qui m'ont apporté leur aide et m'ont soutenu pendant cette thèse. Benoit, puis Camille, en particulier m'ont hébergé à Paris à chaque fois que j'en ai eu besoin. Mes collègues de bureau, et ceux qui en sont partis depuis que j'ai commencé cette thèse, ont été les compagnons d'innombrables discussions à bâtons rompus, de parties de billards et de pintes. Je remercie mes parents qui m'ont toujours poussé à aller plus avant et m'ont témoigné un soutien sans faille dans tout ce que j'ai entrepris, sur le plan des études, de ma vie professionnelle ou personnelle. Enfin, je remercie Dolly, dont la présence à mes côtés est précisément tout ce qu'il m'a toujours fallu. Outre ma compagne, c'est l'intellect avec lequel je peux échanger le plus directement. J'aime à constater combien nos travaux de recherche respectifs s'en ressentent.

Table des matières

Résumé (en français) & Abstract (in english)

Résumé

Nous cherchons à améliorer l'état de l'art des implémentations de contrôle de flux d'information dans les systèmes Linux. Le contrôle de flux d'information vise à surveiller la façon dont l'information se dissémine dans le système une fois hors de son conteneur d'origine, contrairement au contrôle d'accès qui ne peut permettre d'appliquer des règles que sur la manière dont les conteneurs sont accédés.

La première contribution de cette thèse est un plugin pour le compilateur GCC permettant d'extraire les graphes de flot de contrôle des fonctions du noyau. Ces graphes servent de support à l'analyse statique comme à la visualisation.

Ensuite, la question s'est posée de savoir si le framework des Linux Security Modules qui est utilisé pour implémenter les moniteurs de flux d'information est implémenté de telle sorte à permettre la capture de tous les flux produits par les appels système. Nous avons créé et implémenté une analyse statique permettant de répondre à ce problème. Cette analyse statique, dont la correction a été prouvée avec Coq, nous a permis d'étendre le framework LSM pour capturer tous les flux.

Enfin, nous avons constaté que les moniteurs de flux actuels n'étaient pas résistants aux conditions de concurrence entre les flux et ne pouvaient pas traiter certains canaux ouverts tels que les projections de fichiers en mémoire et les segments de mémoire partagée entre processus. Nous avons implémenté Rfblare, un nouvel algorithme de suivi de flux, formellement prouvé avec Coq.

Abstract

We look forward to improving the implementations of information flow control mechanisms in Linux Operating Systems. Information Flow Control aims at monitoring how information disseminates in a system once it is out of its original container, unlike access control which can merely apply rule on how the containers are accessed.

The first contribution of this thesis is a plugin for the GCC compiler able to extract the control flow graphs of the Linux kernel functions. These graphs can be used as a basis for static analyses and visualizations.

Secondly, we studied the Linux Security Modules framework which is used to implement information flow trackers to answer an open problem: is the framework implemented in such a way that all flows generated by system calls can be captured? We have created and implemented static analysis to address this problem and proved

its correction with the Coq proof assistant system. This analysis has allowed us to improve the LSM framework in order to capture all flows.

Finally, we have noted that current information flow trackers are vulnerable to race conditions between flows and are unable to cover some overt channels of information such as files mapping to memory and shared memory segments between processes. We have implemented Rfblare, a new algorithm of flow tracking. The correction of this algorithm has been proved with Coq.

A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.

Frank Herbert, *Dune*, 1965.

Chapitre 1

Introduction

De nombreuses thèses portant sur la sécurité informatique commencent par rappeler comment, à présent que nous vivons dans la société de l'information, nous devenons de plus en plus incapables de nous passer des outils informatiques et comment les menaces associées deviennent de plus en plus sophistiquées, ce qui appelle des contremesures appropriées. En 2017, ces introductions sont réellement surannées et n'effraient plus personne. Néanmoins, n'ayant pas beaucoup plus d'imagination qu'un autre, je viens de faire exactement le même type d'accroche, quoique de manière détournée.

Les travaux que nous avons menés dans cette thèse portent donc sur la sécurité de l'information, et plus précisément sur un contexte, celui des systèmes d'exploitation utilisant un noyau Linux et deux propriétés de sécurité à garantir durant toute la vie du système.

la confidentialité Seuls les utilisateurs autorisés doivent pouvoir accéder aux informations jugées sensibles (que cette classification émane de la Loi, d'un administrateur système sourcilleux ou d'un autre utilisateur soucieux de sa vie privée). Si un utilisateur parvient à lire le contenu d'un fichier privé d'un autre utilisateur par exemple grâce à la complicité d'un tiers, si un pirate parvient à exfiltrer le dernier film encore en post-production d'un studio ou encore si un médecin envoie via sa messagerie personnelle des dossiers de patients, il s'agit d'une violation, respectivement, de la vie privée de l'utilisateur, du secret industriel du studio et du secret médical du patient.

l'intégrité Les informations stockées dans le système ne doivent pas subir de modifications leur faisant perdre leur valeur aux yeux du système ou des utilisateurs qui en sont les propriétaires. L'écrasement accidentel d'un manuscrit de thèse en cours, l'introduction de transactions datant de 2016 dans le livre de comptes de 2017 ou encore la création non autorisée d'un compte utilisateur dans un service par un pirate sont trois exemples de corruptions intolérables, touchant respectivement le dossier dans lequel était contenu le manuscrit, le livre de compte et la base de données des utilisateurs.

Pour répondre aux besoins de confidentialité et d'intégrité, les administrateurs système définissent des *politiques de sécurité* attribuant à chaque utilisateur des *permissions* et à chaque objet du système susceptible de contenir des informations un *niveau de sécurité*. Ces politiques expriment des besoins de sécurité mais ne suffisent pas à les

satisfaire sans les mécanismes permettant leur mise en œuvre effective. La présente thèse porte sur l'un de ces mécanismes : le suivi de flux d'information.

Le moyen le plus répandu de répondre aux objectifs de sécurité est le contrôle d'accès. Pour cela, on associe à chaque niveau de sécurité les permissions qui sont nécessaires pour lire ou modifier les objets assignés à ce niveau. De la sorte, on s'assure que seuls les utilisateurs dûment autorisés peuvent prendre connaissance ou altérer des informations lorsqu'elles sont rangées à l'intérieur d'un fichier, ou de tout autre objet pouvant stocker de l'information, marqué du niveau de sécurité approprié. Cependant, toute la fragilité de l'approche réside dans ce point : une fois les informations hors de leur conteneur d'origine, la politique ne peut plus les protéger. Pour protéger complètement l'information, la politique doit donc être transitive : si Alice a accès à un fichier et Bob non, Alice ne doit pas avoir de moyen de communiquer avec Bob, sinon elle pourrait lui transmettre, même accidentellement, des données du fichier. Cela contreviendrait à la propriété de confidentialité du système.

Le contrôle de flux d'information répond à ce problème. En gardant la trace des mouvements d'information qui ont eu lieu entre les objets dans l'histoire du système, on est en mesure de protéger l'information même une fois en-dehors de son conteneur d'origine, sans restreindre autant les communications que dans le cas du contrôle d'accès. Dans l'exemple précédent, Alice a le droit de communiquer avec Bob, jusqu'à ce qu'elle prenne connaissance du contenu du fichier, après quoi elle doit cesser de communiquer avec Bob afin de ne pas faire fuiter l'information secrète.

Naturellement, les mécanismes de contrôle de flux d'information sont nettement plus compliqués à implémenter et il n'existe pas, à l'heure actuelle, de solution définitive à ce problème. Dans le cadre de cette thèse, nous nous sommes penchés sur les moyens d'implémenter le suivi de flux d'information sous Linux. Le suivi de flux d'information est une étape cruciale du contrôle de flux d'information : il permet de maintenir l'historique des flux dans le système. Les outils de suivi de flux d'information que nous avons étudiés sont tous implémentés dans le noyau Linux en utilisant le *framework* Linux Security Modules (LSM)*. Cependant, ce framework a été conçu à l'origine pour l'implémentation du contrôle d'accès et non de flux, et la question se pose donc de savoir s'il est approprié pour cette tâche. De plus, quand bien même un contrôleur de flux d'information pourrait intercepter tous les flux individuels dans le système, il a encore à traiter un problème supplémentaire : les conditions de concurrence entre les flux peuvent masquer certains d'entre eux aux yeux du moniteur de flux. Nous proposons dans cette thèse le moyen de vérifier par des méthodes formelles une condition nécessaire sur le suivi de flux : pouvoir contrôler tous les flux individuels dans le système ; ainsi qu'une condition suffisante : étant donné une certaine liste de flux individuels suivis, pouvoir observer toute composition de ceux-ci, même en présence de conditions de concurrence.

L'organisation de cette thèse est la suivante. Le chapitre 2 fait un état de l'art du contrôle de flux d'information dans les systèmes d'exploitation et du moyen de l'implémenter ainsi que des moyens d'analyser et de vérifier des propriétés sur le noyau Linux et en particulier le *framework* des *Linux Security Modules*. Le chapitre 3 expose en détail la notion d'« objet du système pouvant contenir des informations » que nous avons délibérément laissée vague dans cette introduction. Nous présentons quels sont les différents conteneurs d'information du noyau Linux et comment les abstractions telles que les fichiers, les *sockets* réseaux, les mémoires partagées ou encore les processus sont implémentées par des structures de données internes au

*. Les termes suivis d'un astérisque à leur première occurrence sont définis dans le glossaire page 189.

noyau. Dans le chapitre 4, nous présentons notre première contribution qui n'est pas directement en lien avec le suivi de flux d'information mais l'analyse du noyau Linux. En effet, nous avons attaqué dans cette thèse des défis scientifiques, à l'aide de méthodes formelles et d'outils théoriques, et nous avons également fait face à des problèmes techniques d'ingénierie logicielle. La base de code du noyau Linux est particulièrement vaste, complexe et évolue rapidement. Notre première tâche a donc été de comprendre son organisation, de prendre en main son code et de proposer des modèles pour la décrire, l'analyser et la visualiser. Le chapitre 4 consiste en la description du projet Kayrebt, une suite d'outils pour extraire du noyau du système d'exploitation des graphes d'appels et de flot de contrôle ainsi qu'une interface graphique de visualisation. Dans le chapitre 5, nous présentons notre seconde contribution, traitant de la condition nécessaire exprimée plus haut. Nous listons les appels système du noyau Linux et nous proposons une analyse statique pour vérifier que le framework LSM permet bien à un moniteur de flux d'information implémenté à l'aide de celui-ci d'observer les flux causés par ces appels système. Enfin, dans le chapitre 6, nous présentons notre troisième contribution, une réponse à la condition suffisante exprimée plus haut sous la forme d'un nouvel algorithme de suivi de flux accomplissant deux objectifs : résister aux conditions de concurrence et suivre les flux dus aux projections de fichiers et aux mémoires partagées, des conteneurs d'information qui ne sont, à l'heure actuelle, traités convenablement par aucun moniteur de flux d'information implémenté pour Linux. Nous montrons premièrement la vulnérabilité des moniteurs de flux implémentés pour Linux en exposant plusieurs attaques exploitant le même défaut contre tous ces moniteurs, pourtant développés de manière indépendante. La correction de notre nouvel algorithme est démontrée avec l'assistant à la preuve Coq. Nous montrons la praticité de notre approche en implémentant notre nouvel algorithme sous la forme de *Rfblare*, un nouveau module LSM. Les attaques que nous avons développées pour effectuer des flux d'information échappant aux autres moniteurs sont inopérantes contre *Rfblare*.

Chapitre 2

État de l'art

En étudiant la littérature portant sur le contrôle de flux d'information dans les systèmes UNIX, et Linux en particulier, nous sommes tombés sur des implémentations s'étendant sur toute la période de ces cinquante dernières années. Les principes du contrôle de flux d'information sont en effet très étudiés mais les modalités de son implémentation sont toujours un objet de recherche active. Dans le cadre particulier du noyau Linux, nous nous sommes ensuite penchés sur les moyens et les risques de l'interposition dans les appels système pour garantir des propriétés de sécurité, et dans notre cas, suivre des flux d'information. Enfin, comme nous nous sommes intéressés aux moyens de construire formellement des implémentations correctes de moniteurs de flux pour Linux, nous avons étudié la très riche histoire des analyses statiques et dynamiques ayant pour objet la vérification de propriétés du noyau.

2.1 Contrôle d'accès et contrôle de flux d'information

Le contrôle de flux d'information est un mécanisme de sécurité consistant à surveiller comment l'information se propage dans un système, quels sont les flux d'information qui y ont lieu, afin d'avoir une vision précise de ce que chaque conteneur d'information contient. Cette information permet de classer chaque conteneur à un certain niveau de sécurité et de mettre en application des politiques de sécurité visant à contrôler l'accès à l'information d'utilisateurs du système en fonction de leur accréditation. Selon le contexte, les notions de *système*, de *conteneur d'information* et de *flux d'information* peuvent varier largement.

Le contrôle de flux d'information dans les systèmes d'exploitation est né d'un besoin de formaliser les objectifs de sécurité des systèmes, et de proposer des politiques pour atteindre ces objectifs. Le contrôle d'accès obligatoire a constitué un thème de recherche important à partir de la fin des années 1960 lorsque le *Department of Defense (DoD)*[*] états-unien a commencé à chercher des moyens permettant d'appliquer les mêmes classifications et garanties de confidentialité dans les systèmes informatiques que dans le reste de ses archives et communications.

Ces besoins ont inspiré la *System Development Corporation* qui a développé ADEPT-50, un système comportant un nouveau modèle de sécurité connu par la suite comme *High Water Mark*, et exposé dans un article de Weissman [102]. Ce système considère quatre types d'objets : les utilisateurs, les fichiers, les terminaux et les sessions. Chaque

objet est associé à un *profil de sécurité* composé d'une *autorité*, d'un certain niveau de sensibilité pris dans une échelle de valeurs, d'un ensemble de *catégories* qui sont une partition non-hiérarchique des différents thèmes d'information contenue dans le système et d'une *concession* qui est l'ensemble des utilisateurs privilégiés pouvant accéder à cet objet. La concession est définie comme :

- lorsqu'il s'agit d'un utilisateur, cet utilisateur lui-même par convention ;
- lorsqu'il s'agit d'un terminal, l'ensemble des utilisateurs pouvant s'y connecter ;
- lorsqu'il s'agit d'une session, l'utilisateur l'ayant initiée ;
- et enfin, lorsqu'il s'agit d'un fichier, l'ensemble des utilisateurs autorisés à le consulter selon l'accès demandé.

ADEPT-50 considère deux types d'accès, lecture seule et lecture–écriture, et le mécanisme de contrôle de flux comprend aussi, particularité notable, le mécanisme de verrouillage : plusieurs lecteurs peuvent accéder à un même fichier simultanément, mais les écrivains obtiennent un accès exclusif.

Le travail de Weissman donne ainsi une définition formelle aux concepts de *classification* et de *permission* utilisés libéralement par le DoD dans ses normes et procédures : la *classification* d'un objet est le profil de sécurité minimal nécessaire pour accéder à cet objet et la *permission* est le profil de sécurité maximal que peut endosser un utilisateur. Ces définitions ont été reprises dans tous les modèles de sécurité inspirés par le DoD. L'élément intéressant dans ce modèle est la distinction faite entre l'utilisateur et sa session. Tout utilisateur a un profil de sécurité fixe, dépendant de ses attributions. Cependant, lorsqu'il se connecte au système, sa session n'est pas ouverte avec ses droits maximaux, mais au contraire avec le plus petit profil de sécurité satisfaisant le terminal auquel il est connecté (tous les terminaux ne permettent pas d'accéder aux mêmes informations du système et de lancer les mêmes commandes). Par la suite, à chaque action faite dans la session, si le profil de sécurité de celle-ci est inférieure aux permissions requises, il est augmenté jusqu'au niveau minimal permettant de satisfaire l'accès, dans la limite du niveau de l'utilisateur. Lorsqu'un fichier est créé ou modifié, son profil ne devient pas celui de l'utilisateur mais celui de la session. Ceci permet de maintenir les informations à un état le plus « public » possible et de ne pas restreindre indûment l'accès à l'information. C'est de ce mécanisme que vient le nom de *high water mark*, qui évoque la hauteur maximale atteint par un cours d'eau lors d'une crue. La marque des objets modifiés devient la plus haute marque des objets consultés précédemment durant la session. Ainsi, si un général se connecte au terminal de la cantine pour commander un sandwich, sa commande pourra être satisfaite par le cantinier de service. En revanche, s'il le fait depuis le terminal de son bureau sur lequel il a consulté un rapport sur l'arsenal nucléaire soviétique (ou nord-coréen, au XXI^{e} siècle), seul son loyal aide de camp sera suffisamment habilité pour lui apporter son plateau-repas.

Sensiblement à la même période, Bell et LaPadula ont proposé un modèle de contrôle d'accès multi-niveaux [52]. Dans ce modèle, les processus (ou *sujets*) tout comme les *objets* du système (essentiellement des fichiers mais le modèle est volontairement vague sur ce point) ont un niveau de sensibilité. Dans le cas des processus, il s'agit de l'accréditation de leur propriétaire. Le modèle décrit deux types d'actions des sujets sur les objets : la lecture et l'écriture. Un processus ne peut lire un objet que si son accréditation est supérieure ou égale à la sensibilité de l'objet ; inversement, il ne peut y écrire que si son accréditation est inférieure ou égale. En appliquant cette politique, il est clair qu'aucun processus d'accréditation basse ne peut accéder à une

information provenant d'un objet de sensibilité haute, même en collaborant avec un « traître » de niveau haut. La notion de contrôle de flux d'information est donc présente. Contrairement à la politique de *high water mark*, le niveau d'un sujet ou d'un objet ne peut pas changer : une action est soit autorisée soit interdite mais elle ne modifie le niveau d'aucun sujet ni objet.

Les travaux fondateurs de Denning [20] sont les premiers à introduire explicitement la notion de flux d'information, et à formaliser un modèle des politiques de contrôle de ces flux. La première contribution de Denning est une relation *can flow* entre classes de sécurité. Les classes de sécurité sont une partition abstraite des objets d'un système. Dans le cas du modèle de Bell–LaPadula, il s'agirait par exemple des sensibilités des objets. La relation *can flow* décrit quels sont les flux d'information autorisés dans le système. Deux classes sont en relation s'il est permis de copier de l'information d'un objet de la première classe dans un objet de la seconde. Cette relation est en fait un préordre :

— Elle est réflexive, de sorte qu'il est toujours permis qu'un conteneur *conserve* l'information qu'il possède déjà.

— Elle est transitive, car sinon, il serait possible de réaliser des flux illégaux en composant des flux légaux.

Le modèle original ajoute de plus l'anti-symmétrie, transformant donc le préordre en ordre partiel, en arguant que si deux classes d'information peuvent librement échanger de l'information alors une des deux classes est redondante. Toutes ces conditions — ajoutées à l'hypothèse raisonnable que dans un système il y a un nombre fini de classes de sécurité, pour des raisons pratiques d'administration — font qu'il est possible de décrire les politiques sous la forme de treillis*. Dans ce modèle formel, il devient possible d'exprimer de nombreuses politiques pré-existantes dans un formalisme commun et de les comparer. L'argumentaire de l'article s'appuie sur le présupposé que les mécanismes censés mettre en application les politiques de sécurité ne peuvent tenir compte que de la classe des objets concernés lors d'un accès, qu'il s'agisse de l'autoriser ou de l'interdire. En particulier, il n'est pas considéré que les flux passés du système soient pris en compte, sauf en changeant définitivement la classe de sécurité des objets pour restreindre les flux possibles. C'est précisément cet élément qui est remis en cause par les mécanismes de traçage de flux d'information, dont l'intuition transparaît dans ADEPT-50 [102].

Les travaux de Jaume et ses collaborateurs, en particulier, portent sur les différences entre le contrôle d'accès et le contrôle de flux d'information [44–46]. Dans la théorie qu'il développe, Jaume présente le contrôle d'accès comme une propriété sur les *états* d'un système tandis que le contrôle de flux porte sur les *exécutions* dans le système. Plus concrètement, le contrôle d'accès promet qu'il n'est pas possible de se trouver dans un état où une information provenant d'une source d'une certaine classe de sécurité A se trouve dans un conteneur d'information d'une classe B alors qu'aucun sujet ayant la permission de lire du contenu d'une source A n'a la permission de modifier une destination B, et inversement, qu'aucun sujet ayant la permission de modifier un conteneur de B n'a le droit de lire depuis une source A. Dans le cas du contrôle de flux d'information, il est garanti que rien de ce que les sujets ne peuvent faire ne peut causer de flux d'une classe A vers une classe B si ce flux est interdit par la politique. Comme on le voit, les motivations sont similaires mais l'implémentation est différente. Les politiques de contrôle d'accès construites sur le modèle de Denning garantissent automatiquement le contrôle de flux d'information [20]. En revanche, il est possible de

garantir le contrôle de flux autrement que par le contrôle d'accès, et ce, de manière strictement plus fine.

En effet, on peut considérer par exemple la matrice de droits d'accès suivants, extraite de « Flow based interpretation of access control » [46].

	o_1	o_2	o_3	o_4
Alice	read, write		read	
Bob	read	read,write		
Charlie		read,write		write

Supposons que l'on associe à chaque conteneur o_i de cet exemple la classe de sécurité C_{o_i} et à Alice, Bob et Charlie (ou mieux dit, aux processus lancés par ces utilisateurs) respectivement les classes A, B et C. On peut dériver du tableau la relation suivante indiquant les accès permis.

$$
\begin{array}{lll}
C_{o_1} \to A & A \to C_{o_1} & C_{o_3} \to A \\
C_{o_1} \to B & C_{o_2} \to B & B \to C_{o_2} \\
C_{o_2} \to C & C \to C_{o_2} & C \to C_{o_4} \\
A \to A & B \to B & C \to C \\
C_{o_1} \to C_{o_1} & C_{o_2} \to C_{o_2} & \\
C_{o_3} \to C_{o_3} & C_{o_4} \to C_{o_4} &
\end{array}
$$

Cette relation est naturellement close par la réflexivité car il est impossible de faire en sorte qu'un conteneur ne puisse pas contenir son propre contenu. On constate en revanche que cette relation n'est pas transitive. On n'a pas par exemple $A \to C_{o_2}$ alors qu'Alice est en mesure d'écrire dans le fichier o_1 que Bob peut ensuite lire pour en copier le contenu dans o_2. Dans le modèle de DENNING, cette politique est incohérente car $\langle \{A, B, C, C_{o_1}, C_{o_2}, C_{o_3}, C_{o_4}\}, \to \rangle$ ne constitue pas un ensemble partiellement ordonné. Considérons à présent les deux exécutions suivantes qui ne diffèrent que par l'ordre des instructions exécutées.

1. *Alice* `reads from` o_3	1. *Bob* `reads from` o_1
2. *Alice* `writes to` o_1	2. *Alice* `reads from` o_3
3. *Bob* `reads from` o_1	3. *Alice* `writes to` o_1

Dans les deux cas, tous les accès individuels sont autorisés. La première exécution présente un flux illégal de C_{o_3} vers B via A puis C_{o_1}, et donc tout modèle de sécurité correct se doit de l'empêcher, mais la deuxième ne comporte que des flux légaux. Or, la deuxième exécution n'est permise par aucune politique de type DENNING car, pour être transitive, elle obligerait à avoir $C_{o_3} \not\to B$, bien qu'il n'y ait aucun flux de C_{o_3} à B dans *cette* exécution. Les mécanismes s'appuyant sur le contrôle d'accès sont donc trop restrictifs dans le sens où ils interdisent des classes entières d'exécutions ne causant aucun flux illégal mais impossibles dans des politiques transitives. Les mécanismes de traçage de flux d'information visent à résoudre ce problème en distinguant les deux cas d'exécution ci-dessus. En effet, on constate qu'ici, ce qui pose problème est le fait que le flux o_3 vers o_1 via Alice a lieu *avant* le flux o_1 vers Bob, alors que dans la seconde exécution, il a lieu *après*, et n'influence donc pas ce que Bob lit depuis o_1. Il est donc suffisant pour un mécanisme de sécurité de maintenir pour chaque conteneur

d'information un historique des flux passés ayant influencé son contenu pour pouvoir distinguer les exécutions comportant des flux illégaux de celles n'en comportant pas.

Il existe en réalité deux types de contrôle d'accès, discrétionnaire et obligatoire. Les politiques de contrôle d'accès discrétionnaire consistent en une matrice telle que présentée plus haut. Chaque utilisateur administre complètement et en totale liberté les droits d'accès de ses propres fichiers et autres conteneurs. Le contrôle d'accès obligatoire en revanche limite la liberté des utilisateurs et permet à l'administrateur du système — parfois connu comme l'officier de sécurité dans la littérature — de poser des restrictions d'accès irrévocables sur certains conteneurs. Les utilisateurs peuvent ajouter des restrictions supplémentaires mais ne peuvent pas retirer les restrictions posées par l'administrateur. Il est donc possible pour l'administrateur dans le cadre du contrôle d'accès obligatoire d'appliquer des politiques fortes, et entre autres de garantir la transitivité de la politique.

La principale différence entre le contrôle d'accès obligatoire et le contrôle de flux est que le contrôle d'accès est sans état. Il ne peut garantir l'absence de flux illégaux qu'en restreignant les politiques applicables. En revanche, le contrôle de flux d'information peut appliquer plus de politiques tout en maintenant les garanties d'absence de flux illégaux, au prix d'un état à maintenir. Par exemple, le contrôle de flux d'information permet à un utilisateur de prendre connaissance de données d'un certain niveau de sensibilité sans lui permettre de les diffuser à son tour, tandis que le contrôle d'accès permettrait soit d'autoriser l'accès et la diffusion, soit de les interdire tous les deux. Le contrôle d'accès discrétionnaire, quant à lui, reviendrait à appliquer des politiques non-transitives et sans maintenir d'historique sur les flux passés. Il ne permet que l'application de politiques simples et échoue à garantir l'absence de flux illégaux dans le cas général. Il suffit par exemple à un processus non autorisé de corrompre ou tromper un processus autorisé pour exfiltrer des données confidentielles.

2.2 Modèles de politiques et de traçage de flux

Comme expliqué plus haut, appliquer une politique de contrôle des flux d'information plus générale que celles permises par le contrôle d'accès nécessite de maintenir un historique des flux passés dans le système. Cependant, cet historique n'a pas besoin d'être un détail complet de tous les flux individuels ayant eu lieu. Il suffit de savoir, pour chaque conteneur d'information, quelles sont les classes de sécurité ayant participé à au moins un flux à destination de ce conteneur. Cela permet de savoir quelles sortes d'informations sont susceptibles de fuiter lors d'un flux depuis ce conteneur. Il s'agit donc d'abstraire le flux.

Propagation de teinte

La manière de loin de la plus répandue de maintenir cette connaissance est la *propagation de teinte*, ou *taint tracking* en anglais. Cette approche consiste à attacher à chaque conteneur d'information un *label*, une méta-donnée indiquant quelles classes de sécurité ont influencé le contenu courant du conteneur. Ce label est propagé en même temps que les données lorsqu'un flux a lieu d'un conteneur à un autre, et le label du conteneur destination est mis à jour pour refléter le fait que son contenu a été modifié par la source. Le terme de *propagation de teinte* vient de ce mécanisme. Tout se passe comme si les conteneurs étaient marqués d'un coup de peinture et que les conteneurs en participant à des flux d'information se « tachaient ». À tout instant de la

vie du système, la couleur des tâches d'un conteneur permet de savoir qui l'a touché. Ce concept de marquage remonte au moins aux travaux séminaux de Fenton [29]. En réalité, Fenton a exposé l'idée de la propagation tout en l'écartant dans son premier modèle. Les marques associées aux conteneurs, les registres d'un modèle abstrait de machine dans son cas, sont statiques. Les mérites de la propagation ont donc été redécouverts plus tard, notamment par McIlroy et Reeds [56] comme détaillé en section 2.3.

Le concept de propagation de teinte est très générique et a été adapté pour de nombreux usages. On peut citer notamment le langage Perl qui possède un mécanisme permettant de marquer les données provenant de sources non sûres (les entrées de l'utilisateur, essentiellement) et d'empêcher qu'elles ne soient évaluées comme du code, ou comme une requête de base de données [70, 80].

Terminologie

Tous les auteurs ont introduit une terminologie similaire mais néanmoins subtilement différente pour se référer aux concepts du contrôle de flux d'information. Dans cet état de l'art, nous ne prétendons pas unifier toutes les terminologies et notations mais nous avons malgré tout simplifié le discours en utilisant le même mot pour le même concept tant que faire se pouvait. Voici un petit glossaire de ces termes.

sécurité La sécurité est une propriété abstraite d'un système auquel chaque modèle associe une définition formelle. On lui associe classiquement les sous-propriétés de confidentialité, d'intégrité et de disponibilité [79]. Dans le cadre du contrôle de flux d'information, seules la confidentialité et l'intégrité sont discutées dans les modèles.

confidentialité La confidentialité représente l'impossibilité pour tout utilisateur d'un système d'accéder à des données qu'il n'a pas l'autorisation de consulter. Nous ne considérons ici que les violations actives de la confidentialité, impliquant un attaquant.

intégrité L'intégrité, pour ce qui concerne le contrôle de flux d'information, représente l'impossibilité pour tout utilisateur d'un système d'influencer, altérer ou supprimer des données qu'il n'est pas autorisé à modifier. De la même manière que précédemment, nous ne considérons pas les atteintes accidentelles à l'information, comme le changement de valeur d'un bit dû à un rayonnement cosmique malencontreux.

processus Un processus est une entité active du système, exécutant du code. Selon les cas, il peut s'agir d'un unique fil d'exécution (*thread**) ou bien d'un processus à la mode UNIX (essentiellement, un groupe de *threads* selon les modèles mais ceci constitue un détail qui ne sera pertinent dans cette thèse qu'à partir du chapitre 3).

objet Un objet est une abstraction du système qui peut contenir de l'information et peut être la source ou la destination d'un flux. Nous utilisons également le terme de conteneur d'information.

label Un label est une méta-donnée attachée à chaque objet indiquant quel est son niveau de confidentialité ou d'intégrité, ou comportant de quelque manière que ce soit l'information suffisante pour permettre au mécanisme de contrôle de flux d'information de décider de la légalité des flux d'information.

tag Présents dans certains modèles, les tags identifient chacun une catégorie ou une source primitive d'information du système. En soi, ils ne représentent pas un

niveau de confidentialité ou de sécurité, mais ils composent les labels, auxquels les modèles donnent une sémantique en termes de niveaux de confidentialité et intégrité.

déclassification La *déclassification* est l'opération consistant à diminuer le niveau de sécurité d'un objet. Augmenter le niveau de confidentialité d'un objet n'entraîne pas de risque concernant la confidentialité, cela ne fait que rendre l'objet moins accessible. En revanche, la déclassification, qui est l'opération inverse, pose un risque. Si l'objet contient des données très sensibles, elles peuvent être exposées à des individus non autorisés, il s'agit donc d'une opération nécessitant des privilèges.

approbation L'*approbation*, *endorsement* en anglais, est le pendant de la déclassification pour ce qui est de l'intégrité. Ce n'est pas un risque pour la sécurité de diminuer l'intégrité d'un objet, mais en revanche augmenter l'intégrité est une opération privilégiée car si l'objet contient des données corrompues ou qui ne sont pas de confiance, elles risquent de contaminer d'autres objets de confiance. C'est cette opération que l'on appelle l'approbation.

2.2.1 Plusieurs niveaux d'application du traçage

Le contrôle de flux d'information peut s'appliquer à de nombreux étages dans un système d'exploitation, voire entre systèmes d'exploitation en réseau. En effet, selon ce que l'on considère être un *conteneur d'information*, et selon les flux que l'on souhaite observer, on peut distinguer plusieurs niveaux de granularité.

niveau matériel Les flux sont tracés dans tout le système avec l'aide du matériel, par exemple directement dans le processeur. Les flux sont causés par des instructions machine et les conteneurs d'information sont les registres du processeur et les mots mémoires. Par exemple, une instruction telle que `ADD EAX,EBX`, qui signifierait "ajouter dans le registre `EAX` le contenu du registre `EBX`", produirait un flux de `EBX` à `EAX`. seL4, Panorama, RIFLE ou encore Loki appliquent ce type de traçage de très bas niveau [62, 99, 108, 111].

niveau programme Les flux sont tracés à l'intérieur d'un processus, ou d'un groupe de processus comme la machine virtuelle Java. Les flux sont causés par des instructions du langage de programmation et les conteneurs sont les variables manipulées par le programme ainsi que ses entrées–sorties, comme les descripteurs de fichiers qu'il manipule. Quelques exemples célèbres d'implémentations sont JFlow ou Jif [64, 66], le travail de Genaim et Spoto [32] et JBlare [40] pour le langage Java, FlowCaml [72] pour ML.

niveau système d'exploitation Les flux sont tracés par le noyau dans tous les processus hors du noyau. Les flux sont causés par des appels systèmes et les conteneurs sont des abstractions proposées par le système d'exploitation comme les fichiers, les processus, les sockets réseaux, etc. Les exemples d'implémentation sont très nombreux dans cette catégorie [10, 16, 33, 51, 67, 76, 100, 109] et nous les décrivons dans la section 2.3.

On peut bien entendu imaginer des niveaux supplémentaires où à l'intérieur d'un réseau, les ordinateurs, bases de données, montages réseaux, etc. constituent les conteneurs et où le traçage des flux est effectué par le réseau. Hauser a par exemple décrit un mécanisme permettant de propager des teintes entre machines dans les paquets *Internet Protocol* (IP)* en exploitant une extension du protocole [38]. DStar [110] et

Aeolus [10] appliquent également du contrôle de flux d'information dans des systèmes distribués. Il est également courant que le traçage soit effectué dans plusieurs niveaux à la fois, par exemple à l'intérieur de la machine virtuelle Java et au niveau du système d'exploitation, à travers des mécanismes de coopérations. TaintDroid met ainsi en œuvre plusieurs niveaux de traçage intra- et inter-processus et compte sur le noyau pour assurer la persistance de son traçage de flux [28]. De la même manière, le mécanisme de traçage des flux de Panorama est implémenté dans le matériel mais Panorama reconstruit la vue sémantique du noyau pour donner plus de contexte aux flux [108]. Enfin, DroidScope analyse des logiciels malveillants en les faisant tourner dans une machine virtuelle et son analyseur de teintes travaille également au niveau matériel, chaque instruction du processeur émulé participant à la propagation. Il reconstruit la vue sémantique à la fois du noyau (Android, en l'occurrence) et de la machine virtuelle Dalvik (faisant tourner les applications Android) pour donner un sens aux flux produits par le logiciel étudié. Il peut ainsi tracer les flux entre objets Java des applications Android [106].

2.2.2 Contrôle de flux d'information décentralisé

Le contrôle de flux décentralisé, ou *Decentralized Information Flow Control (DIFC)*, permet à chaque utilisateur du système de choisir lui-même la politique s'appliquant aux données lui appartenant, c'est-à-dire aux données qu'il a injectées dans le système. Le DIFC a été créé par Myers et Liskov [65, 66]. La décentralisation est effective et aisée dans un système de contrôle d'accès discrétionnaire, puisque chaque fichier ou processus appartient à un utilisateur et cet utilisateur est en mesure d'accorder le droit de lecture ou d'écriture à d'autres utilisateurs. Le système ne fait *qu'appliquer* la politique choisie par l'utilisateur (qui est cependant limité à leurs propres conteneurs, les utilisateurs ne contrôlent pas ce qu'il advient de leur information une fois dans le conteneur d'un autre utilisateur). À l'inverse, dans les systèmes de contrôle d'accès obligatoire, le super-administrateur met des restrictions très fortes sur les permissions possibles pour chaque objet afin de garantir des politiques fortes. De plus, de nombreux systèmes comme Bell–LaPadula font l'hypothèse de la *stabilité*, c'est-à-dire que les conteneurs ne sont pas censés changer de niveau de sensibilité. D'autre part, la déclassification (baisser le niveau de sensibilité d'une donnée après l'avoir auditée) ou au contraire l'approbation (augmenter le niveau d'intégrité d'une donnée après l'avoir vérifiée) ne sont pas des opérations conçues pour être simples dans le système car la moindre erreur peut conduire à des fuites d'information ou des corruptions et réduire l'utilité du mécanisme de sécurité à néant. Les utilisateurs n'ont donc aucun contrôle sur leurs propres données. Dans un système simpliste de contrôle de flux d'information, la situation est similaire. Les utilisateurs doivent compter sur l'administrateur pour appliquer des labels corrects sur leurs propres fichiers, et surtout pour les modifier, et cela n'est pas sans difficulté.

Dans un DIFC, la notion de *propriétaire* est donc très importante car elle constitue une délégation des droits d'administration. En termes de contrôle de flux d'information, cela signifie le droit de modifier voire de retirer la partie du label contribué par l'utilisateur à un conteneur d'information donné. Imaginons par exemple un système très abstrait avec trois utilisateurs : Alice, Bob et Charlie. Alice souhaite envoyer un mot doux à Bob sans que celui-ci (un brin vantard) ne puisse le montrer à Charlie. Un moyen simple de le faire est de modifier le label du message pour y incorporer une certaine marque et ensuite de demander au système d'appliquer une simple politique indiquant « Bob peut lire les conteneurs possédant cette marque », « Charlie NE peut

PAS lire les conteneurs possédant cette marque ». De la sorte, Bob, en téléchargeant le message et en l'enregistrant dans un fichier verra son fichier marqué. Il sera en mesure de le lire, mais pas de le montrer à Charlie. Plus tard, si Alice le désire, elle pourra lever cette restriction en mettant simplement à jour la politique, ou bien en retirant la marque du fichier.

Dans cette section, nous avons donc établi les principes fondateurs du contrôle de flux d'information, les différences par rapport au contrôle d'accès et les modalités de sa mise en œuvre, mais nous n'avons pas discuté des moyens de l'implémenter concrètement dans un système d'exploitation. La présente thèse porte en particulier sur l'implémentation, et la vérification de l'implémentation, d'un moniteur de flux d'information dans le noyau Linux. Dans la section suivante, nous allons par conséquent étudier l'historique des implémentations de contrôle de flux d'information ainsi que les principes et inventions qui ont guidé le développement de ce domaine de recherche, avant de nous concentrer sur les implémentations dédiées à Linux.

2.3 Implémentations de contrôle de flux d'information dans les systèmes d'exploitation par propagation de teinte

Dans cette section, nous nous intéressons plus particulièrement aux implémentations de contrôle de flux d'information par propagation de teinte à l'échelle des systèmes d'exploitation. Cette catégorie comprend à la fois des systèmes d'exploitation entiers [56], des noyaux de systèmes d'exploitation [100, 109], des composants de sécurité pour ces noyaux [17, 33, 51, 67, 76], des moniteurs implémentés comme des programmes surveillant les flux depuis l'extérieur du noyau de tout ou partie des processus [28, 51] ou encore des *Application Programming Interfaces (API)** permettant le développement d'applications distribuées contrôlant les flux entre les processus de ces applications [10]. Le seul critère retenu est que les conteneurs d'information considérés sont de la granularité du système d'exploitation (processus, fichiers, Inter-Process Communication (IPC)* divers, etc.) et que les flux sont caractérisés par des appels des processus au système d'exploitation.

Il faut noter qu'il n'y a pas vraiment de barrières étanches entre les différents niveaux d'application (matériel, langage, système, etc.) du contrôle de flux. En effet, les travaux de Denning [20, 21] et de Myers et Liskov [64, 66] présentent des modèles très généraux qui sont ensuite appliqués au cas particulier des langages de programmation mais les modèles de systèmes d'exploitation détaillés plus bas y font référence néanmoins [51, 100, 109]. Les distinctions que nous avons introduites entre les expressions « contrôle de flux d'information », « contrôle d'accès obligatoire » et « propagation de teinte » ne sont pas non plus présentes dans tous les articles et travaux. Nous rappelons donc ici que la différence fondamentale qui est intéressante dans la présente thèse est le fait que le contrôle de flux d'information nécessite de maintenir un état sur les flux passés s'étant produits dans le système. Dans le cas où cet état prend la forme d'un label attaché aux objets (y compris les processus du système) et évoluant au fur et à mesure que des flux se produisent, nous considèrons qu'il s'agit de propagation de teinte.

2.3.1 Implémentations de systèmes d'exploitation sur mesure

IX

Les premiers prototypes de systèmes d'exploitation centrés autour du contrôle de flux d'information ont été construits sur mesure dans ce but précis, afin de démontrer l'implémentabilité et les avantages de ce type de solution. Ils réutilisent des portions de systèmes pré-existants, réécrits en profondeur pour intégrer le mécanisme de propagation de teintes. Le premier système d'exploitation à avoir implémenté le contrôle de flux d'information, à notre connaissance, est IX par McIlroy et Reeds en 1992 [56]. Ce système est une variante du système d'exploitation UNIX supportant une politique de sécurité forte, à plusieurs niveaux, dans la tradition des préconisations de l'*Orange Book* du DoD [53]. McIlroy et Reeds décrivent leur système comme appliquant du contrôle d'accès obligatoire. En réalité, d'après nos définitions données plus tôt il s'agit bien de contrôle de flux d'information. En effet, plusieurs éléments distinguent clairement IX du modèle de Bell–LaPadula ou même du *high water mark*. En premier lieu, IX abandonne la dualité sujet–objet et ne considère plus que des conteneurs d'information impliqués dans des flux (ou dans leurs propres termes « *places between which data occasionally flows* »). Deuxièmement, les auteurs font le constat que contrairement au contrôle d'accès où seules les ouvertures de fichiers ont besoin d'être contrôlées, le contrôle de flux requiert d'appliquer la politique à chaque opération de lecture et écriture car les labels peuvent être modifiés à chaque flux.

McIlroy et Reeds expliquent également les problèmes liés au fait que le contrôle de flux crée lui-même des canaux cachés de communication. L'exemple donné par les auteurs est celui d'un processus avec un niveau d'autorisation bas voulant obtenir des données d'un processus collaborant, mais de niveau bien supérieur. Une méthode simple à mettre en œuvre et relativement fiable est pour le processus haut d'écrire ou non dans un certain fichier de niveau de sensibilité bas initialement. S'il écrit, cela augmente automatiquement le niveau de sensibilité du fichier. Le processus bas peut facilement observer le niveau du fichier en tentant de le lire : si la lecture échoue, c'est que le niveau de sensibilité était haut. De la sorte, le processus haut peut donc communiquer un bit d'information au processus bas. L'opération peut être « industrialisée » en la répétant à intervalles réguliers (et en remplaçant le fichier pour le ramener au niveau bas) afin de transmettre n'importe quelle donnée, un bit à la fois. Enfin, l'article évoque aussi le problème de l'*explosion de teintes*, qui est le nom donné au phénomène faisant que dans un système appliquant du contrôle de flux d'information, au fil du temps, les conteneurs tendent à acquérir des labels de plus en plus restrictifs, et que les possibilités de flux d'information deviennent donc de plus en plus restreintes. Le problème est similaire a celui d'un programme qui allouerait toujours plus de mémoire sans jamais la libérer. Cela met en danger la disponibilité du système, qui est un objectif de sécurité.

Asbestos

Asbestos est un autre exemple de nouveau système d'exploitation conçu en vue de la protection des données et de l'isolation des services. Ses créateurs, VanDeBogart et al., prêchent pour l'exploration de nouvelles possibilités dans le contexte non restreint d'un nouveau système d'exploitation [100]. Une des idées nouvelles constituant une avancée majeure est la décentralisation des politiques de flux d'information. En effet, Asbestos propose non seulement une politique globale permettant de donner des garanties classiques comme celles du DoD mais aussi la possibilité pour chaque application de définir sa politique d'isolation et de protection de ses données vis-à-vis

des autres processus du système. Enfin, Asbestos propose, pour la première fois à notre connaissance, un moyen de lutter contre l'explosion de teintes par un mécanisme connu dans le contexte des bases de données sous le nom de *poly-instanciation* [55]. On se référera avec profit à l'article « Solutions to the Polyinstanciation Problem » [43] pour une présentation détaillée de ce mécanisme dans son contexte original. La poly-instanciation consiste à séparer les états ou sessions qu'un serveur maintient avec différents clients. D'ordinaire, le seul but est celui de l'isolation, afin de garantir la confidentialité des données de chaque processus vis-à-vis des autres. Dans un système à propagation de teintes, ce mécanisme prévient également la contamination par les teintes qui empêcherait le serveur de s'adresser à plus d'un client. L'alternative pourrait évidemment être le lancement dynamique d'un serveur différent pour chaque arrivée d'un nouveau client mais cela aurait un coût important en termes de ressources. La poly-instanciation, si elle est supportée par le système d'exploitation qui garantit une isolation correcte entre les différentes sessions d'un même processus, fournit une solution à la fois plus légère, plus flexible et plus robuste.

Les labels d'Asbestos sont organisés autour de la notion de tags, qui sont des identifiants associés à certaines classes de contenus. Les tags sont alloués par les utilisateurs du système et administrés chacun par son allocateur. Les labels sont simplement définis comme un ensemble de tags, provenant de différents utilisateurs. Lorsqu'un flux d'information survient d'un conteneur vers un autre, les tags de la source sont ajoutés dans ceux de la destination. L'allocation d'un tag confère automatiquement le privilège d'ajouter ce tag au label de n'importe quel conteneur possédé par l'utilisateur et de le retirer du label de n'importe quel conteneur du système (naturellement, si un utilisateur quelconque avait le droit de marquer n'importe quel conteneur avec son tag, il pourrait empêcher son propriétaire légitime d'y accéder). Ce modèle confère donc à chaque utilisateur le droit de définir une politique sur ses propres données, tout en étant contraint par les politiques définies par les autres utilisateurs quant aux tags qui ne lui appartiennent pas. Le système est l'arbitre garant de la bonne application des politiques. Asbestos est donc une implémentation de DIFC dans un système d'exploitation [65].

Les « nouvelles primitives » qu'Asbestos promet se résument à un élément central : le passage de message. Les messages sont considérés comme des objets à part entière du système et obéissent donc aux règles présentées plus haut. Pour empêcher que ces règles ne créent des canaux cachés, Asbestos rend ces interfaces *non fiables*. Si l'envoi d'un message échoue à cause de permissions insuffisantes et que le révéler exposerait de l'information sur les labels du *destinataire*, alors l'erreur n'est pas signalée à l'émetteur. Ceci cause bien sûr des problèmes car il est malaisé pour un développeur d'applications de ne pouvoir compter que sur des canaux non fiables, y compris localement — il est plus habituel de considérer uniquement les communications de machine à machine non fiables. Pour comprendre quel canal caché est évité par ce mécanisme, on peut considérer deux processus, un processus H pouvant lire du contenu marqué du tag t mais n'ayant pas la possibilité de le déclassifier et un processus L ne pouvant pas le lire. Supposons que le processus L souhaite lire un fichier marqué t, avec la complicité du processus H. H peut allouer un nouveau tag t' puis créer un port de communication qu'il marque ou non d'intégrité haute avec son nouveau tag. Ensuite, L tente d'envoyer un message à ce port. Si le port n'a pas été marqué avec t', le message de L pourra arriver, en revanche si le port a été marqué, comme L ne possède pas le niveau d'intégrité suffisante, L n'a pas le droit d'envoyer ce message. Si l'erreur est rapportée à L, selon que l'envoi échoue ou non, L sait si H a tagué ou non son port, ce qui permet à H d'envoyer passivement un bit d'information à L sans communiquer directement

avec lui. H peut ainsi envoyer à L le fichier que ce dernier n'a pas le droit de lire, bit par bit.

HiStar

HiStar est un système d'exploitation inspiré directement d'Asbestos mais mettant l'accent sur des inventions différentes [109]. HiStar réutilise les tags d'Asbestos mais requiert que tous les flux soient explicitement demandés. C'est-à-dire que contrairement à Asbestos qui utilise un modèle de *label flottants*, où ce sont les flux d'information qui causent automatiquement un changement des labels des processus concernés, HiStar utilise un modèle d'*élévation explicite* des labels. Le processus désireux d'être la destination d'un certain flux d'information doit auparavant élever son label à un niveau suffisant pour dominer le label de la source de ce flux. Ce parti-pris ferme un canal caché. Tout comme Asbestos, HiStar applique un contrôle de flux d'information décentralisé, le processus qui crée un tag étant libre de l'ajouter et de l'enlever de tous les objets lui appartenant. Grâce à ce mécanisme, HiStar peut se passer de super-utilisateur. Toutefois, étant donné que le modèle de labels est très abstrait et général, l'administration d'un tel système est compliquée. HiStar donne plusieurs exemples de politiques à appliquer à des logiciels particuliers mais la composition des politiques appliquées à différents composants au sein d'un même système n'est pas discutée. Il semble difficile en particulier de démontrer la cohérence d'une politique de sécurité, surtout si l'on considère le défi de sa maintenance dans le temps.

Un autre point de ressemblance avec Asbestos est le fait qu'HiStar définit de nouvelles primitives de communications, certaines présentant des ressemblances avec celles de certains systèmes UNIX mais demeurant uniques dans leur fonctionnement. En premier lieu, HiStar généralise le concept de *répertoire* des systèmes traditionnels en celui de *conteneur*. Un conteneur est donc une sorte d'annuaire référençant l'existence d'autres objets, pouvant eux-mêmes être des conteneurs. Les conteneurs portent un label et cela permet donc de cacher l'existence de certains objets à des processus non-autorisés. Les processus autorisés peuvent, eux, consulter le conteneur pour connaître les labels des objets qui y sont référencés et savoir à quel point ils doivent élever leur label pour y accéder. Les segments et espaces d'adressage sont aussi des objets gérés par HiStar. Tout processus possède un certain espace d'adressage qui représente sa mémoire virtuelle, composée d'un ensemble de segments, chacun avec un label. Ce système permet à un *thread* d'écrire dans la mémoire d'un autre de manière sûre. Cependant, il n'est pas clair que ce mécanisme ne puisse pas être contourné. En effet, si un processus possède deux segments en lecture–écriture, si l'un des deux a un label de confidentialité plus fort que l'autre, le processus peut tout de même écrire du premier dans le deuxième sans faire d'appel système (les lectures ou écritures dans la mémoire se font entièrement en espace utilisateur, dès lors que les segments sont chargés en mémoire), et donc sans qu'HiStar ne puisse voir cette déclassification illégale. Ce point n'est pas discuté dans l'article « Making Information Flow Explicit in HiStar » [109]. Enfin, HiStar définit les *portes* qui sont des fonctions appelables dans un espace d'adressage par un *thread* même s'il ne possède pas cet espace d'adressage, pourvu qu'il en ait connaissance via un conteneur. Lorsqu'un processus passe une porte — ce qui revient à effectuer un appel de fonction appartenant à un autre processus — il peut déléguer certains de ses labels à celle-ci *jusqu'au retour de la fonction*. Ce mécanisme permet par exemple d'implémenter une déclassification contrôlée.

2.3.2 Implémentations dans le cadre de systèmes d'exploitation préexistants

Flume

Flume est un système assez différent de ceux présentés jusqu'ici. Contrairement à Asbestos et HiStar, Flume ne vise pas à protéger l'intégralité du système mais uniquement certaines applications qu'il « encapsule ». En effet, Flume n'est pas implémenté comme un module de sécurité dans le noyau mais comme un moniteur d'exécution en espace utilisateur, tournant dans un système Linux ou OpenBSD. Sous Linux, Flume nécessite cependant quelques ajouts au noyau, sous la forme d'un module de sécurité Linux (voir la section 2.4.2). Les applications surveillées nécessitent également un effort de portage léger pour tourner à l'intérieur de Flume. Néanmoins, cet effort est bien plus léger que celui consistant à réécrire les applications pour leur faire bénéficier des nouvelles primitives des systèmes Asbestos et HiStar. En particulier, Flume conserve les IPC de Linux (tuyaux et sockets réseaux en particulier). Flume met l'accent sur la simplicité de concevoir des politiques de sécurité en structurant les labels en plusieurs parties distinctes : des parties différentes gèrent respectivement la confidentialité, l'intégrité et les privilèges de déclassification et d'approbation. Flume présente enfin le compromis principal justifiant le découpage de cette section en « systèmes sur mesure » d'une part et « systèmes intégrés à Linux » d'autre part : un système sur mesure comme Asbestos a une étendue de code à certifier plus petite et un meilleur contrôle sur les canaux cachés, en revanche, un mécanisme de sécurité intégré à Linux est plus susceptible d'être adopté et effectivement utilisé car il bénéficie de la dynamique de développement de Linux et des applications déjà supportées par ce noyau.

Le modèle de label de Flume hérite de celui d'Asbestos la notion de tag comme identifiant opaque d'une certaine classe d'information. Flume est un DIFC, les processus peuvent donc allouer des tags librement et administrer les tags qu'ils allouent eux-mêmes. Les objets surveillés par Flume ont deux labels : un pour la confidentialité, un pour l'intégrité. Les tags sont également séparés en tags de confidentialité et tags d'intégrité. Un label est un ensemble de tags et la sémantique intuitive en est très simple. Si un fichier est marqué d'un tag de confidentialité c alors seuls les processus possédant ce tag peuvent le lire. De la même manière, si un fichier est marqué du tag d'intégrité i alors seuls les processus possédant ce tag peuvent y écrire. À chaque tag t, Flume attache de plus deux *capacités* t^+ et t^-. t^+ représente le droit d'ajouter t à son label (soit de confidentialité soit d'intégrité, selon t) tandis que t^- représente le droit de retirer t. Ces capacités permettent d'implémenter les privilèges de déclassification et d'approbation. Tout comme HiStar, les changements de labels d'un processus p ne peuvent être faits que par p lui-même et doivent être faits explicitement. Les flux n'augmentent pas automatiquement les labels. De plus, les objets passifs du système (les objets n'exécutant pas de code, c'est-à-dire tous exceptés les processus) ont des labels fixes. Un processus créant un objet est libre de choisir ses labels avec la restriction qu'il doit être capable d'y écrire (ce qui empêche un processus très secret de créer un fichier publiquement lisible).

Aeolus

Aeolus [10] est une suite logique aux travaux de Liskov et ses collaborateurs concernant Jflow et son successeur Jif [64, 66] et l'introduction du contrôle de flux d'information décentralisé [65]. Aeolus est une plate-forme permettant de développer des applications distribuées avec des garanties fortes sur les flux d'information vers et

depuis l'application, ainsi qu'entre les *threads* la constituant, qui peuvent être déployés sur plusieurs machines. Aeolus est implémenté comme une bibliothèque Java avec laquelle il est possible de programmer des applications sécurisées. Tout comme les auteurs de Flume, les auteurs d'Aeolus font le constat qu'il est primordial de fournir des primitives de communications, de gestions des labels et de déclassification/approbation simples pour permettre l'adoption du contrôle de flux d'information par le plus large public possible et que seule la décentralisation du contrôle de flux permet suffisamment de souplesse. Contrairement à Flume, toutefois, les privilèges de déclassification et approbation ne sont pas exercés à travers des *capacités* mais par des *autorités*, un concept jugé plus usuel pour les administrateurs par les auteurs.

Tous les objets d'Aeolus et tous les processus (qui ne sont pas des processus du système d'exploitation mais des processus légers de la machine virtuelle Java dont le lancement et l'exécution sont contrôlés par Aeolus) possèdent deux labels, un pour la confidentialité, l'autre pour l'intégrité, ainsi qu'un propriétaire. Ce propriétaire n'est pas directement un utilisateur mais plutôt un rôle ; ce qui permet aux utilisateurs de cloisonner leurs activités. Comme dans Flume, les labels sont composés de tags, qui peuvent être créés par des processus pour catégoriser leurs informations. Lorsqu'un processus crée un tag, son propriétaire acquiert automatiquement *autorité* sur ce tag. L'autorité permet d'exercer les privilèges de déclassification et approbation. L'autorité pour un tag peut également être déléguée et révoquée à d'autres propriétaires. La hiérarchie des propriétaires constitue un graphe orienté acyclique qui permet de savoir qui administre quoi. De plus, amélioration notable par rapport aux précédentes approches, Aeolus permet de définir des groupes de tags qui peuvent être manipulés comme une entité unique, et simplifient l'expression des politiques.

Aeolus définit plusieurs primitives de communication. En premier lieu, les processus peuvent utiliser le Remote Procedure Call (RPC)* pour appeler une fonction d'un processus depuis un autre. Ce qui est intéressant dans ce mécanisme est qu'il permet aux processus de définir dans quelles conditions leurs fonctions peuvent être appelées. En particulier, il est possible d'attacher un propriétaire avec des autorités spécifiques à chaque fonction appelable à distance. Cela permet à un processus par exemple d'appeler une méthode de déclassification sur des données lorsqu'elle n'a pas elle-même l'autorité nécessaire. Un cas pratique pourrait par exemple être la fonction d'authentification. Les utilisateurs ne souhaitant pas que leur mot de passe soit divulgué à une application tout entière lorsqu'ils s'authentifient peuvent auditer une courte et simple fonction qui vérifie leur mot de passe et déclassifie le résultat « mot de passe correct » ou « mot de passe incorrect ». Ils vérifient ensuite que le propriétaire de cette fonction ne réalise aucune autre action dans le système et confie à ce propriétaire l'autorité sur le tag *mot_de_passe*. Par la suite, il leur suffit de toujours envoyer leur mot de passe dans un message tagué avec *mot_de_passe* pour avoir la garantie qu'il est impossible à l'application de faire quoi que ce soit de leur mot de passe à part appeler la fonction auditée. Aeolus propose également un mécanisme de mémoire partagée entre processus reposant sur les mêmes règles, ainsi qu'un système de fichiers permettant la persistance des labels sur les fichiers. Enfin, Aeolus permet aux applications de communiquer avec l'extérieur à travers des socket réseaux ou autres mécanismes fournies par le langage Java. En cas de réception d'une donnée externe, le processus « à la frontière » doit avoir un label d'intégrité vide et en cas d'émission, un label de confidentialité vide, afin d'éviter tout flux non autorisé qui contournerait les protections d'Aeolus.

2.3.3 Implémentations dans le noyau Linux

Il existe plusieurs exemples d'implémentations de solutions de sécurité plus ou moins complètes utilisant le contrôle de flux d'information à l'intérieur du noyau de systèmes d'exploitation commercialisés. La plupart ciblent le noyau Linux. Ce dernier comprend depuis 2001 une interface pour implémenter des modules de sécurité appelée *Linux Security Modules* [104, 105]. L'usage et le fonctionnement de LSM sont détaillés en section 2.4.2.

Laminar

Les auteurs de *Laminar*, Roy et al., font le constat de la complémentarité entre les contrôleurs de flux d'information au niveau langage de programmation et au niveau système d'exploitation : tandis que le niveau langage permet de tracer les flux à une échelle très fine, le niveau système d'exploitation permet de tracer les flux dans l'intégralité du système [76]. Laminar propose donc un mécanisme implémenté à la fois dans le noyau Linux et dans la machine virtuelle Java afin de tirer le meilleur parti des deux niveaux de traçage. Contrairement à la plupart des approches précédentes, Laminar ne cherche pas à définir de nouveaux canaux de communication mais au contraire à réutiliser les abstractions existantes de Linux, héritées d'UNIX.

Laminar hérite principalement son modèle de labels de Flume. Les conteneurs de flux d'information sont les *threads*, qui sont la plus petite unité ordonnançable par le noyau, et les fichiers au sens large, c'est-à-dire tous les objets manipulées via un descripteur de fichier (voir le chapitre 3 pour une présentation détaillée de ces concepts). Tout comme dans Flume, les labels des objets sont deux ensembles de tags, l'un pour la confidentialité, l'autre pour l'intégrité. Logiquement, un tag est utilisé de manière exclusive soit pour la confidentialité, soit pour l'intégrité. L'ensemble des parties de l'ensemble des tags forme naturellement un treillis et les objets ne comportant pas de label se distinguent par leur deux ensembles vides. Les threads possèdent en plus un ensemble de capacités. Étant donné un tag t, t^+ est la capacité d'ajouter ce tag à leur label (dans le cas d'un tag de confidentialité, cela leur donne le droit d'accéder aux contenus teintés par t ; dans le cas de l'intégrité, cela correspond au privilège d'approbation) et t^- est la capacité de retirer ce tag de leur label (dans le cas d'un tag d'intégrité, cela leur donne le droit de lire depuis des sources moins fiables ; dans le cas de la confidentialité, cela correspond au privilège de déclassification).

Laminar peut faire appliquer toutes les politiques classiques proposées par les modèles précédents mais impose des limitations sur la définition de flux : un flux est un déplacement d'information d'un objet à un autre, dont l'un au moins est un *thread*. De plus, suivant l'exemple d'HiStar, les *threads* doivent changer explicitement leurs labels de manière appropriée avant qu'un flux puisse avoir lieu. Les flux ne les font pas évoluer automatiquement.

En plus d'appliquer le contrôle de flux d'information à l'échelle du système, Laminar bénéficie d'une granularité plus fine pour les programmes en Java grâce à son implémentation dans la machine virtuelle Java. La partie Java de Laminar peut raffiner l'analyse en contrôlant les flux d'information entre les *threads* des programmes Java et les objets (au sens de structures de données) qu'ils allouent dynamiquement. La partie Java de Laminar ne permet pas la manipulation des tags en permanence. Au contraire, les zones où des tags peuvent être ajoutés ou enlevés sont restreintes à des *zones de sécurité* qui doivent être définies par le programmeur. Chaque zone porte des labels et des capacités qui deviennent temporairement celles du *thread* qui y entre. Un *thread*

ne peut accéder à des données portant un label que s'il se trouve à l'intérieur d'une zone de sécurité. D'après les concepteurs de Laminar, Roy et al., ce parti-pris réduit la quantité de code qu'il est nécessaire d'auditer pour garantir les bonnes propriétés de sécurité d'un logiciel. Néanmoins, on constate quelques défauts dans cette approche. D'une part, il est clair que Laminar ne peut pas être utilisé tel quel pour protéger les applications existantes. En effet, Laminar rajoute plusieurs appels système pour la manipulation des tags. Cela constitue une modification profonde du noyau, qui complique sa maintenabilité. De plus, cela signifie que les applications ne peuvent pas être portées directement sur un système Laminar. D'autre part, bien que le modèle de la partie Java de Laminar soit très puissant, là encore, la nouvelle interface de Laminar et l'usage de régions de sécurité impliquent un effort important pour adapter les anciennes applications dans ce nouveau système.

Blare

Blare est un prototype de moniteur de flux d'information développé à l'origine par Zimmermann [113, 114] puis Hiet [40] puis Hauser [38]. Blare se distingue des autres approches présentées ici par plusieurs aspects. En premier lieu, l'objectif initial de Blare n'est pas, comme pour les autres outils, l'application d'une politique de contrôle de flux d'information mais un outil de *détection d'intrusion* basé sur une politique de flux. La différence semble mineure d'un point de vue conceptuel puisque faire appliquer une politique et en surveiller les violations semble revenir au même. Cependant, dans la pratique, cela a conduit la conception et le développement de Blare à des choix atypiques. En premier lieu, Blare ne s'oppose pas au déroulement normal des appels système. Il est inséré dans l'exécution des appels système mais se contente de lever une alerte lorsqu'il détecte une possible violation de la politique de flux d'information sans renvoyer d'erreur. Blare comportait à l'origine un *daemon** en espace utilisateur pour vérifier la légalité des flux d'information. Dans les versions les plus récentes, cependant, Blare est implémenté entièrement dans le noyau. L'implémentation repose sur le framework LSM. Comme pour Laminar, les conteneurs d'information comprennent les *threads* ordonnancés par le noyau et les fichiers (incluant les tuyaux et les sockets réseaux). Blare surveille de plus les files de messages System V et considère aussi les projections en mémoire et segments de mémoire partagés entre processus. Sur ce dernier point, l'implémentation est défaillante cependant, comme expliqué en section 6.1.2. Dans le chapitre 6, nous présentons une contribution résolvant le problème de manière prouvée.

Le modèle de Blare est également radicalement différent des approches précédentes. Il s'agit du modèle conçu par Viet Triem Tong, Clark et Mé [101]. Les labels sont toujours des éléments de l'ensemble des parties des tags, organisé comme un treillis. Cependant, chaque objet n'a qu'un seul label et les tags ne sont pas classés en tags de confidentialité et tags d'intégrité. Chaque tag caractérise en fait une source primaire d'une certaine information. Chaque objet possède deux labels : un label d'information et un label de politique. Le label d'information est un ensemble de tags. Il définit quelles sont les informations que le conteneur est susceptible de contenir à tout instant en représentant l'accumulation des sources d'information qui l'ont teinté. Le label de politique est un ensemble d'ensembles de tags. Un conteneur est dans un état légal si son label d'information est un sous-ensemble d'au moins un des ensembles de son label de politique. Cela permet de définir des politiques à la fois de confidentialité et d'intégrité très simplement. Les tags qui ne sont présents dans aucun des ensemble du label de politique d'un objet sont tout bonnement inaccessibles à cet objet, ce qui capturent l'as-

pect « confidentialité », et certains mélanges d'informations de provenances différentes sont interdits, alors même que les informations sont individuellement permises, ce qui capture l'aspect « intégrité ». On peut par exemple permettre qu'un fichier comptable contiennent des factures de l'année 2016 ou de l'année 2017, mais pas des deux à la fois, en donnant au fichier le label de politique $\{\{factures_{2016}\}, \{factures_{2017}\}\}$. La propagation se fait de manière naturelle à chaque occurrence d'un flux d'information, en changeant le label d'information de l'objet destination par l'union des labels d'information des objets source et destination. Si le nouveau label d'information de la destination n'est pas légal, une alerte est levée. Contrairement à toutes les implémentations étudiées jusqu'ici, aucune distinction n'est faite dans Blare entre les *threads* ou processus et les autres objets, ils portant tous un label de politique et le modèle prévoit que des flux puisse avoir lieu d'un objet à un autre même lorsqu'aucun des deux n'est un *thread*. Plusieurs appels système de Linux permettent par exemple à un flux d'avoir lieu entre deux fichiers, sans que les informations ne transitent par un processus. De plus, contrairement aux approches tels que HiStar et Laminar, on notera que les labels de Blare sont flottants. De manière générale, les concepteurs de Blare sont plus attachés à couvrir le plus de canaux ouverts possibles et n'ont que peu considéré le problème des canaux cachés, y compris ceux créés par leur propre outil.

Le modèle de Blare a été décliné en plusieurs implémentations au-delà du noyau Linux. L'implémentation construite avec LSM pour le noyau Linux *vanilla** a été renommée *KBlare* [33, 38–40, 115]. Il existe également une version pour la machine virtuelle Java [40] nommée *JBlare*. La coopération est possible entre KBlare et JBlare, afin de raffiner la propagation de teinte effectuée par KBlare. Le mécanisme semblable de Laminar a été développé concurremment et indépendamment de celui de Blare. Enfin, plus récemment, une version spécifique pour les système Android, nommé *AndroBlare*, a été développée par Andriatsimandefitra. La principale différence par rapport à KBlare est que les systèmes Android disposent d'une IPC privilégiée pour la communication entre applications : le *binder*, qui est implémenté dans le noyau. De plus, les applications sont particulièrement bien isolées et ont beaucoup moins de privilèges par défaut que n'en ont des processus lambda sur un système Linux classique. La précision de la propagation de teinte s'en trouve améliorée d'autant.

Weir

Le mécanisme de contrôle de flux d'information *Weir*, conçu et implémenté par Nadkarni et al. [67], est le prototype le plus récent que nous ayons rencontré dans la littérature, ayant été publié en 2016. Comme AndroBlare, il s'agit d'un moniteur de contrôle de flux d'information pour systèmes Android, s'appuyant sur LSM. Weir utilise le modèle de labels de Flume mais contrairement à ce dernier, il utilise des labels flottants, en arguant de leur réalisme par rapport au modèle de changement explicite de labels promu par HiStar. En effet, le changement explicite requiert d'une certaine manière de savoir à l'avance pour son destinataire quel flux va avoir lieu avant de pouvoir l'autoriser, ce qui est aisé dans le cas de la lecture d'un fichier, beaucoup moins dans celui de la réception d'un message depuis un autre processus. Weir fait une contribution intéressante pour limiter l'explosion de teintes inévitable dans un système à labels flottants. Un service utilisé par d'autres processus par exemple se retrouve teinté par tous ceux-ci, et teinte à son tour tous ceux qui l'utilisent. Au bout d'un certain temps, il est inévitable que les flux légaux deviennent de plus en plus difficiles à réaliser et le système devient rapidement inutilisable. Pour pallier ce problème, Weir utilise la poly-instanciation. Comme Asbestos qui maintient des

sessions séparées pour les différents utilisateurs, Weir peut dédupliquer à la volée des processus ou applications du système afin de contrôler à la fois l'empreinte mémoire et la charge CPU des applications et la propagation des teintes. Une nouvelle instance d'une application est lancée chaque fois que cette application est appelée depuis un contexte de sécurité (un certain label) différent. À l'inverse une même application faisant appel deux fois à un même service sera servie par la même instance, qui aura donc gardé l'état de la communication. Ce modèle est particulièrement pertinent sous Android où les applications sont très interconnectées entre elles sans même se connaître. Sous Android, si une application nécessite d'envoyer un courriel, c'est le système qui consulte la liste des applications disponibles pouvant rendre ce service et demande à l'utilisateur de faire son choix parmi les différentes applications disponibles. Une même application est donc susceptible d'être sollicitée depuis des contextes de sécurité très différents et il est important que le mécanisme de poly-instanciation soit transparent du point de vue des applications qui demandent le service. Enfin, Weir met un accent tout particulier sur la commodité et la rétro-compatibilité afin de pouvoir être intégré dans un système existant, où les anciennes applications qui ne créent pas de tags et n'utilisent pas le DIFC continuent à fonctionner.

2.3.4 Autres usages du traçage de flux d'information

Jusqu'ici ont été présentés uniquement des systèmes de contrôle de flux d'information ayant pour but de surveiller un système en activité et d'avertir ou d'empêcher les flux illégaux. Cependant, bien qu'il s'agisse de l'usage historique de la propagation de teinte, ce n'est pas le seul et des travaux récents ont montré que les mêmes outils pouvaient être appliqués dans d'autres contextes comme l'analyse de logiciels malveillants. C'est l'approche appliquée par DroidScope [106] et AndroBlare [1, 2].

2.3.5 Conclusion sur les implémentations de contrôle de flux d'information dans les systèmes d'exploitation

Le domaine des contrôleurs de flux d'informtion est très riche et beaucoup de recherches différentes ont été consacrées au problème de la meilleure implémentation possible. En réalité, la conception d'un contrôleur de flux d'information implique de faire de nombreux compromis.

- Vaut-il mieux implémenter un nouveau système ou greffer un mécanisme de sécurité sur un système existant ? La première solution permet de mieux contrôler les canaux de communications dont disposent les processus mais la deuxième favorise la maintenabilité et l'adoption effective de la solution proposée, à terme.
- Vaut-il mieux couvrir le plus possible de canaux existants, ou juste restreindre les possibilités d'IPC offerts aux processus ? Là encore, la première solution semble plus sûre mais la deuxième est plus réaliste ; les canaux existants sont tous utilisés par au moins une application, qu'il faudra réimplémenter si ce canal est bloqué.
- Est-il valable de changer la sémantique des IPC existants si cela permet d'éviter des flux illégaux ? Asbestos et ses descendants, typiquement, rendent les tuyaux non fiables en ne permettant pas à l'émetteur de savoir si son message a bien pu arriver à destination. Cela clôt un canal caché de communication. Cependant, les applications qui dépendent de la sémantique spécifiée par POSIX des tuyaux ne fonctionneront plus.

	Type de système	Objets considérés	Labels flottants	Étendue du code de confiance	Labels fixes hors processus
IX	Nouveau système d'exploitation	Processus, fichiers	Oui	Noyau	Non
Asbestos	Nouveau système d'exploitation	Processus, fichiers, messages	Oui, initialement	Noyau	Non
HiStar	Nouveau système d'exploitation	Processus, espaces d'adressages, portes, segments, conteneurs	Non	Noyau	Oui
Flume	Processus « encapsuleur » de processus à surveiller	processus, fichiers, tuyaux, sockets	Non	Moniteur d'exécution et le noyau	Oui
Aeolus	Plate-forme d'exécution distribuée	Processus, fichiers, ports de communication externes	Non	Machines virtuelles Java, noyaux, communication entre machines	Oui
Laminar	Module de sécurité Linux et machine virtuelle Java modifiée	Processus, fichiers, sockets	Non	Noyau Linux	Oui
KBlare	Module de sécurité Linux	Processus, fichiers, sockets, mémoires partagées (partiellement), files de messages System V	Oui	Noyau Linux	Non
Weir	Module de sécurité Linux et nouveau service Android pour la gestion des tags	Processus, fichiers, sockets réseaux, binder d'Android	Oui	Noyau Linux	Oui

TABLE 2.1 – Comparatif des principales caractéristiques de certains contrôleurs de flux d'information présentés

— Quelle étendue de code peut-on accepter comme étant de confiance ? Dans tout système d'exploitation, même ceux pourvu d'un mécanisme de sécurité, il est inévitable qu'une certaine partie du code doive être considérée de confiance : le code implémentant le mécanisme de sécurité ou pouvant le supplanter.
— Est-il bénéfique de déporter du code du contrôleur de flux d'information vers l'espace utilisateur ? Le code dans l'espace utilisateur paraît moins bien isolé que le code du noyau, cependant il tourne avec moins de privilèges, ce qui est une bonne chose pour réduire l'impact des vulnérabilités dans son code.

Nous résumons les caractéristiques des différents contrôleurs de flux d'information présentés dans cette section dans la table 2.1.

Contrôler les flux d'information suppose d'être capable de *détecter* tous les flux afin d'être en mesure d'avoir une vue de l'état du système conforme à la réalité. Dans le cadre des systèmes d'exploitation, l'interception des appels système s'est imposée comme la candidate naturelle et idéale pour la détection des flux d'information. Dans la prochaine section, nous détaillons les notions d'espace utilisateur, noyau et appels système et nous étudions les raisons, avantages et inconvénients de l'interception des appels système pour l'implémentation des mécanismes de sécurité à l'échelle du système d'exploitation.

2.4 Interposition dans les appels système

Contrairement à un langage de programmation ou un micro-processeur, il n'y a pas vraiment de liste définitive des « instructions » d'un système d'exploitation. Dans la suite de ce discours, on considére un système d'exploitation abstrait composé d'un *noyau*, pilotant le matériel et orchestrant la vie du système, et d'un *espace utilisateur* comprenant tous les autres programmes, qu'ils appartiennent réellement à des utilisateurs ou bien qu'ils soient des portions du système d'exploitation comme des programmes réalisant des tâches d'administration en arrière-plan. Le noyau et les programmes en espace utilisateur vivent des existences très différentes. Le tableau 2.2 illustre les principales différences entre le noyau et les programmes en espace utilisateur. Il existe une frontière naturelle entre le noyau et l'espace utilisateur constitué par l'interface des *appels système* — terme qui en réalité aurait été mieux traduit comme *appels au système*. En effet, aucune fonction du noyau ne peut être appelée depuis l'espace utilisateur à part les appels système. Si l'on fait l'hypothèse que toutes les opérations provoquant des flux d'information sont implémentées dans le noyau, on peut donc considérer le noyau comme une machine à faire des flux, et l'ensemble des appels système comme son jeu d'instructions. Pour contrôler les flux, il est donc suffisant pour un mécanisme de sécurité de s'interposer entre l'espace utilisateur et le noyau lors des appels système.

2.4.1 Risques inhérents à l'interposition de contrôles de sécurité dans les appels système

Garfinkel a étudié dans « Traps and Pitfalls » [31] les risques que court un mécanisme de sécurité surveillant l'activité de processus utilisateur à la frontière des appels système. Il les classifie en cinq grandes catégories.

Désynchronisation entre l'état maintenu par le mécanisme et l'état du noyau
Parfois, un mécanisme de sécurité a besoin de maintenir un état pour prendre des

TABLE 2.2 – Principales différences entre le noyau et les processus s'exécutant en espace utilisateur

Noyau	Processus en espace utilisateur
Est chargé par le matériel et lancé au démarrage du système.	Est chargé par le noyau et est généralement lancé à la demande de l'utilisateur.
Ne s'arrête pas, l'arrêt inopiné du noyau rend le système inutilisable.	Peut s'arrêter à la fin de sa tâche ou être interrompu par l'utilisateur ou si survient une erreur.
A les privilèges maximaux sur le matériel.	Ne peut accéder à aucun matériel directement, toute communication doit se faire à travers le noyau.
A accès à la mémoire physique.	N'a accès qu'à sa propre mémoire virtuelle, cloisonnée des mémoire virtuelle des autres processus.
Peut exécuter toutes les instructions du processeur.	Est limité à un sous-ensemble sûr d'instructions.
Peut exécuter n'importe quelle action dans le système, transférer de l'information d'un processus à un autre, écrire dans un fichier, etc.	Doit demander les opérations privilégiées au noyau, en effectuant un *appel système*.
Est très sensible aux bugs, la moindre erreur peut compromettre la sécurité de tout le système et n'importe quelle donnée qui y est stockée.	Peut se compromettre soi-même ainsi que les données et programmes du même utilisateur.

décisions de sécurité futures. Par exemple, un contrôleur de flux d'information peut avoir besoin de se souvenir qu'un processus partage une zone de mémoire avec un autre processus pour se souvenir qu'il existe un flux d'information entre les deux qui va durer au-delà de l'appel système ayant servi à mettre en place cette mémoire partagée. Si le mécanisme de sécurité ne parvient pas à maintenir à jour la liste des mémoires partagées, il prendra des décisions de sécurité erronées. De plus, ces erreurs ont tendance à s'accumuler avec le temps, à moins qu'il ne soit possible de se « resynchroniser » avec l'état du noyau, qui fait toujours autorité.

Réplication incorrecte d'une fonctionnalité du noyau Comme le mécanisme de sécurité doit prendre des décisions impliquant les arguments passés en paramètre des appels système, il doit parfois leur appliquer les mêmes transformations que le noyau lui-même. En particulier, lorsqu'un des arguments est un chemin, il est nécessaire d'en calculer la forme canonique. Cet algorithme est notoirement compliqué, entre autres pour des raisons historiques datant d'UNIX. Si le mécanisme de sécurité doit le répliquer plutôt que de réutiliser l'implémentation du noyau, il court le risque de mal interpréter les arguments de l'appel système et donc de prendre des décisions de sécurité erronées. Plus insidieux encore — c'est un point qui n'est pas directement abordé par GARFINKEL, même si l'implémentation du mécanisme de sécurité est plus correcte que celle du

noyau (du point de vue des normes POSIX ou *Single UNIX Specification* par exemple), c'est malgré tout celle du noyau qui fait autorité car c'est lui qui, à la fin, interprétera les arguments et effectuera l'opération sensible dans l'appel système.

Conditions de concurrence dans les arguments des appels système Ce problème ne se présente que dans le cas des mécanismes de sécurité implémentés en espace utilisateur. Dans ce cas, il se présente de la manière suivante :

1. un processus surveillé effectue un appel système, le mécanisme de sécurité est notifié immédiatement ;
2. le mécanisme interrompt l'exécution du processus pour vérifier l'appel système, ses arguments, etc. ;
3. si tout est correct, l'appel système est autorisé, l'exécution du processus reprend ;
4. le processus effectue son appel système.

Le problème de cet enchaînement d'actions est qu'il est possible que les arguments de l'appel système changent entre le moment où ils sont vérifiés par le mécanisme de sécurité et le moment où le noyau exécute l'appel système. Cela peut être le cas par exemple si l'un des arguments est une chaîne de caractère stockée en mémoire partagée. Si le mécanisme de sécurité est implémenté dans le noyau en revanche, le problème ne se pose pas car le noyau commence toujours les appels système par la copie des arguments de l'espace utilisateur vers sa propre mémoire. Cela garantit que les arguments qui sont vérifiés sont bien ceux qui serviront lors du reste de l'exécution de l'appel système.

Protection partielle d'une ressource Si certaines structures de données du noyau peuvent être modifiées par des moyens qui ne sont pas couverts par le mécanisme de sécurité, alors les protections de celui-ci pourront être contournées. Un cas classique présenté par Garfinkel [31] est celui des descripteurs de fichiers qui peuvent être échangés entre processus via un message sur une socket de type UNIX. Cela permet à un processus d'accéder à un fichier ouvert sans l'ouvrir lui-même. Si un mécanisme de sécurité cherche à contrôler les ouvertures de fichiers mais ne contrôle seulement l'appel système `open` et pas les appels système de la famille de `sendmsg` qui permettent de transférer des descripteurs de fichiers, ils ne pourront qu'échouer à appliquer leur protection. Sous Linux, il faut également se méfier des appels système multiplexeurs `ioctl` et `fcntl` qui permettent d'accéder à des fonctions particulières des fichiers et périphériques. Les possibilités offertes par ces appels sont très variables car elles dépendent du matériel et la véritable opération qu'ils effectuent est spécifiée par un des arguments.

Mauvais comportement des applications dû à l'interception des appels système Lorsque les mécanismes de sécurité interceptent les appels système, ils doivent être en mesure de retourner un code d'erreur approprié au processus dans le cas où les permissions de ce dernier sont insuffisantes. Il est important de retourner un code documenté par l'appel système original car le processus risque d'avoir un comportement inattendu s'il reçoit un code d'erreur auquel il n'est pas préparé. D'autre part, si le mécanisme de sécurité restreint l'ensemble des appels système accessibles à un processus, il doit prendre garde à ne pas empêcher les appels systèmes servant à *réduire* les privilèges. Sinon, il est probable que le processus ignorera l'erreur et continuera malgré tout à s'exécuter, avec des privilèges trop élevés.

Bien que Garfinkel n'ait pas considéré particulièrement le cas des moniteurs de flux d'information, ses observations s'appliquent tout à fait à ce cas, en particulier le premier point concernant la désynchronisation entre l'état du mécanisme de contrôle de flux et le noyau.

2.4.2 Le système des *Linux Security Modules*

Le framework *Linux Security Modules* (LSM) [105] est comme son nom l'indique une interface pour implémenter des modules de sécurité en sus du mécanisme de contrôle d'accès discrétionnaire existant par défaut pour le noyau Linux. Il a été introduit en 2001. Il se présente sous deux aspects. D'une part, des attributs supplémentaires ont été ajoutés aux structures de données du noyau dont l'accès doit être restreint. Ces attributs ne sont pas utilisés par le code standard du noyau et les modules de sécurité sont donc libres de les utiliser pour stocker leur état, et dans le cas des mécanismes de propagation de teintes, les labels des conteneurs d'information correspondant aux structures. D'autre part, des fonctions particulières ont été ajoutées et elles sont appelées à des endroits stratégiques du code du noyau, avant des opérations caractérisées comme sensibles du point de vue de la sécurité par les mainteneurs du noyau. Ces opérations peuvent être par exemple la lecture d'un fichier ou l'envoi d'une donnée sur le réseau. Ces fonctions particulières, appelées crochets (*hooks* en anglais) dans la terminologie LSM peuvent être redéfinies par un module de sécurité pour appliquer des contrôles supplémentaires et éventuellement retourner un code d'erreur pour empêcher la réalisation de l'opération sensible. Nous détaillons plus avant la conception de LSM dans le chapitre 5.

Le framework LSM ne constitue donc pas *stricto sensu* une interface pour l'interception des appels système car en réalité les crochets sont implémentés profondément *dans* les appels système. Il y a plusieurs avantages clairs à cela : d'une part, cela permet de factoriser des crochets dans les appels systèmes qui manipulent les mêmes structures de données internes du noyau, et d'autre part, cela permet aux modules de sécurité de profiter du prétraitement des arguments de l'appel système par le noyau. Par exemple, les adresses mémoires ou descripteurs de fichiers invalides sont détectés avant l'interposition, ce qui facilite le travail des développeurs du module. La conception intelligente de LSM permet également de couvrir une partie des risques exposés dans la section 2.4.1 : il n'y pas de risques de conditions de concurrence concernant les arguments des appels système, et comme les LSM font partie de la conception des appels, les valeurs d'erreur qu'ils peuvent retourner sont documentées, ce qui évite toute mauvaise surprise. Les modules bénéficient de toutes les interfaces de programmation du noyau, et peuvent donc consulter l'état du noyau, ce qui peut permettre de limiter le problème de désynchronisation. Le problème de protection partielle des ressources est déporté quant à lui des modules au framework LSM. En effet, ce sont les seuls points où le mécanisme de sécurité peut non seulement agir (c'est-à-dire prévenir une opération incorrecte) mais aussi plus simplement *observer* l'état du système.

On conçoit donc que la position des crochets dans le code est critique pour assurer de bonnes garanties de sécurité. Si un crochet manque avant une opération sensible, alors le mécanisme de sécurité sera impuissant à la détecter, et a fortiori l'empêcher. À l'inverse, trop de crochets nuiraient aux performances du noyau et rendraient très complexe et difficile la maintenance des mécanismes de sécurité. La bonne position des crochets LSM peut être vérifiée de plusieurs manières : par du test, des analyses statiques, dynamiques, du *model-checking*, etc. Dans la section suivante, nous détaillons l'état de l'art concernant l'analyse du code du noyau, pour donner le cadre général

dans lequel s'inscrivent les approches spécifiquement dédiées à l'analyse de la position des crochets LSM.

2.5 Analyse du code du noyau Linux

Les analyses de programmes ont pour but la vérification de certaines propriétés attendues de ceux-ci. On distingue classiquement deux catégories d'analyses, avec leurs avantages et défauts respectifs.

2.5.1 Analyses statiques

Les analyses statiques vérifient des propriétés sur le code source du programme, en fonction de la sémantique déclarée du langage de programmation, sans requérir de compiler le programme vers le langage machine et encore moins de l'exécuter. Les avantages de ce type d'analyse sont les suivants.

Parfaite couverture de tout le code
Tout le code est pris en compte, même les fonctions rarement appelées peuvent être analysées.

Indépendance vis-à-vis des plate-formes d'exécution
Certaines conditions de concurrence peuvent être difficiles à déclencher pendant une exécution. Les analyses statiques s'abstraient de ce genre de problèmes.

Confinement et reproductibilité
Contrairement à une analyse dynamique qui requiert une exécution et qui peut par conséquent être influencée par l'environnement d'exécution, les analyses statiques peuvent être appliquées en parfaite isolation.

2.5.2 Analyses dynamiques

Les analyses dynamiques consistent en une exécution contrôlée du programme, où des données sur le bon respect de certains invariants, sur l'utilisation des ressources du système, sur le temps d'exécution, sur les entrées–sorties, sur les appels système, etc. peuvent être collectées. Ce type d'analyse a des avantages complémentaires à ceux des analyses statiques.

Indépendance vis-à-vis du langage, du compilateur, etc.
Peu de langages de programmation répandus ont une sémantique formelle, la plupart se contentent de normes exprimés en langue naturelle. De ce fait, beaucoup de programmes sont sous-spécifiés. De plus, les compilateurs étant eux-mêmes des logiciels, ils ne sont pas exempts d'erreurs, pas plus que les plate-formes matérielles. Par conséquent, il peut y avoir des différences entre ce qu'exprime le code d'un programme et son exécution effective. Les analyses dynamiques sont immunisées contre ce type de problèmes.

Possibilité de vérifier des propriétés invérifiables statiquement
En vertu du théorème de Rice, toute propriété non-triviale d'un programme est indécidable statiquement dans le cas général [74, Corollaire B] donc les analyses statiques sont contraintes à faire des approximations. Les analyses dynamiques sont capables de vérifier beaucoup plus de propriétés calculables lors d'une exécution.

2.5.3 Outils dédiés à l'analyse

Dans cette section, nous décrivons les outils qui ont déjà été utilisés pour la vérification du noyau Linux, et en particulier les outils intégrés dans la chaîne de compilation et de tests du noyau. Pour un état de l'art plus complet de toute la théorie et les outils relatifs aux analyses statiques ou dynamiques de code tel que le noyau Linux, on consultera avec profit la thèse de doctorat de Slabý [83].

Approches statiques

De nombreux types d'outils ont été utilisés pour la vérification du noyau Linux. La méthode la plus basique et la plus rapide pour trouver des erreurs consiste à chercher des *motifs* dans le code connus pour être symptomatiques de certaines classes de problèmes. Par exemple, cela peut permettre de détecter l'oubli d'une désallocation de mémoire dans une fonction ou la comparaison directe entre un pointeur et un entier. À l'origine, ces recherches de motifs étaient faites par de simples expressions régulières mais naturellement, la grammaire du langage C n'étant pas régulière, cette méthode n'est ni correcte, ni sûre : il n'y a aucune garantie de trouver tous les problèmes, ni de ne trouver que des problèmes dans la sortie de l'outil. C'est pour répondre à ce problème qu'a été développé *Coccinelle* [69]. Coccinelle est un moteur de patchs *sémantiques*, c'est-à-dire qu'au lieu de travailler sur de simples expressions régulières, il comprend la grammaire du C et peut s'appuyer sur quelques connaissances de la sémantique du langage tels que la commutativité de l'addition et la multiplication, les conversions de type, etc. Cela lui permet de reconnaître des motifs et d'appliquer des transformations complexes. Coccinelle est intégré dans les sources du noyau Linux depuis 2010 [97, commit 74425eee71eb44c9f370bd922f72282b69bb0eab].

Linus Torvalds, créateur et mainteneur-en-chef de Linux, a également développé un analyseur sémantique pour le C appelé Sparse [7, 98]. L'objectif initial de Sparse n'est pas de constituer un compilateur mais uniquement un analyseur capable d'extraire des caractéristiques du code étudié, voire d'en proposer des visualisations (sous forme de graphes de flot de contrôle, typiquement). Certaines vérifications statiques sont également facilitées par cet outil. Par exemple, en annotant les fonctions avec l'information des verrous qu'elles prennent et qu'elles libèrent, il est possible de vérifier qu'en tout point d'une fonction, le nombre de verrous occupés ne dépend pas du chemin suivi pour arriver à ce point (ce qui pourrait être le signe que le long de certains chemins d'exécution, tous les verrous pris ne sont pas libérés comme il faut) [96].

MECA [107] et *CQUAL* [30] ont été utilisés pour vérifier la bonne utilisation des pointeurs venant de l'espace utilisateur dans le noyau [47, 107]. En effet, pour des raisons de sécurité, il est critique que le noyau soit parfaitement isolé des processus utilisateurs. Par conséquent, il faut prendre garde, dans le noyau, de ne pas déréférencer de pointeur fourni par les processus utilisateur (par exemple, en tant qu'argument d'appel système) sans vérifier qu'il pointe effectivement vers de la mémoire à laquelle le processus a légitimement accès. Des fonctions particulières, `copy_from_user`, `copy_to_user`, etc. sont disponibles pour déréférencer correctement ces pointeurs. Les travaux de Yang et al. [107] d'une part et Johnson et Wagner [47] d'autre part visent à garantir qu'aucun pointeur de l'espace utilisateur n'est déréférencé autrement que par ces fonctions. MECA s'appuie sur de la propagation de teintes : les pointeurs provenant des processus sont automatiquement marqués comme teintés à l'entrée des appels systèmes et seules les fonctions citées plus haut peuvent retirer la teinte. Déréférencer un pointeur teinté est signalé comme une erreur. L'approche CQUAL, pour sa part,

s'appuie sur la théorie des types. Chaque type pointeur est raffiné en deux catégories : le type pointeur utilisateur et le type pointeur noyau. Les conversions ne sont autorisées que par les fonctions données plus haut. La principale différence entre ces deux approches est que celle de CQUAL est correcte : toutes les manipulations potentiellement mauvaises de pointeurs sont détectées, au prix de nombreux faux positifs (c'est-à-dire de signalements d'erreurs qui n'en sont pas). À l'inverse, MECA ne prétend pas à la correction, il est possible de le tromper avec du code un peu compliqué et donc des erreurs peuvent lui échapper, mais il rapporte beaucoup moins de faux positifs.

Enfin, *BLAST* [4], un autre analyseur statique, a été utilisé pour détecter des erreurs dans la prise et la libération de verrous, pouvant conduire notamment à des interblocages [61]. Cette approche s'est révélée conclusive mais un peu décevante. En effet, à l'époque des travaux de Mühlberg et Lüttgen [61], le support des pointeurs dans BLAST était trop peu développé pour permettre des analyses pertinentes. Cela a conduit les développeurs du projet *Linux Driver Verification* à produire un nouveau module pour améliorer cet aspect de BLAST [81].

Une limite globale de ces approches est que Linux n'est pas écrit en C strictement conforme à la norme, mais en un dialecte du C défini par le compilateur de la suite *Gnu Compilers Collection* [89] (GCC)*. GCC est le compilateur de référence du noyau Linux, et pendant très longtemps a été le seul compilateur capable de compiler l'intégralité du noyau. Certaines extensions au C apportées par GCC sont explicitement permises dans le noyau [77]. De plus, la norme du C est écrite en langue naturelle et comporte des ambiguïtés et laisse certains détails « définis par l'implémentation » (sous-entendu, du compilateur et de la bibliothèque standard). Un même code peut donc correspondre à plusieurs exécutables ayant des comportements différents. Par conséquent, un programme qui dépend d'une autre analyse syntaxique et sémantique que celle de GCC court le risque de ne pas interpréter le code de la bonne manière et de tirer des conclusions fausses ; en dépit de sa preuve de correction qui repose sur l'hypothèse implicite que la sémantique qu'il donne au code est identique à celle de GCC. Depuis 2014, le développement du noyau Linux, dans le cadre du *Kernel Self-Protection Project*, intègre d'ailleurs des greffons [1] à GCC dans les sources du noyau pour en améliorer la sécurité [14], copiant en cela le travail du groupe *PaX* [86]. Parmi les greffons intégrés ou proposés jusqu'à présent dans le noyau, on peut signaler *kernexec* [12, 24] et *structleak* [13, 24]. Le greffon *kernexec* vise à empêcher l'exécution de code appartenant à des processus utilisateur depuis le noyau. Cela pourrait arriver en trompant le noyau et en lui passant une adresse à laquelle sauter dans le code dans l'espace utilisateur au lieu du noyau, ce qui revient à permettre l'exécution de code arbitraire avec les privilèges du noyau. Le greffon *structleak* quant à lui oblige à ce que les structures marquées comme pouvant contenir des adresses en espace utilisateur (et donc susceptibles d'être exposées aux processus utilisateurs) soient initialisées avec des valeurs connues. Si ce n'est pas le cas, il existe un risque qu'une structure non-initialisée révèle le contenu précédent de la mémoire qui lui est allouée, ce qui peut comprendre des données internes au noyau, des pointeurs révélant l'organisation de son espace mémoire (facilitant ainsi l'exploitation de certaines vulnérabilités), etc.

1. Peut-être plus connus sous le nom anglais de *plugins*.

Approches dynamiques

La principale méthode de détection de problèmes dans le noyau Linux reste à cette heure le test. Néanmoins, de nombreuses approches ont été proposées pour vérifier de manière plus extensive et plus systématique le bon comportement du noyau. Une approche assez ancienne et qui connaît un regain de popularité actuellement est le *fuzzing* [59]. Le fuzzing consiste à tester un logiciel de manière extensive en lui fournissant toutes les combinaisons d'entrées et d'évènements possibles, de manière à exhiber des comportements mal prévus, ou des chemins d'exécution rarement empruntés dans le code. Le fuzzing était à l'origine purement aléatoire mais les *fuzzers* les plus modernes font appel à des techniques provenant à la fois de l'analyse statique et de l'apprentissage artificiel afin de rechercher spécifiquement les entrées susceptibles d'être les plus problématiques [8, 9]. Par exemple, ils vont chercher à tester les indices limites de boucles et de tableaux dans les fonctions pour provoquer des débordements, ou bien, ils vont tenter de déclencher des conditions de concurrence dans les codes multi-threadés s'ils détectent qu'une variable est partagée. Le fuzzing retire ainsi du choix des tests le biais humain qui consiste à ne se concentrer que sur les cas d'usage prévus dans la conception originelle du code. Le fuzzing sert de manière assez classique à détecter les conditions faisant planter les logiciels [82] mais peut aussi servir à découvrir des vulnérabilités permettant des attaques de la confidentialité ou de l'intégrité [57]. En effet, certaines entrées malformées peuvent conduire à esquiver certains tests de sécurité et certains modes d'erreurs peuvent, au lieu de le faire planter, mettre le logiciel dans un état tel qu'il sera plus vulnérable à d'autres attaques.

Le fuzzing a été utilisé à de nombreuses reprises pour tester Linux. La liste exhaustive des travaux de cette branche serait hors-sujet ici mais l'on peut citer comme exemple les travaux de Sim, Kuo et Merkel [82] ou de Mendonça et Neves [57]. Concernant les outils dédiés au fuzzing, ils sont également nombreux à s'être succédés et ils sont utilisés intensivement pour tester les nouveaux pilotes de périphériques ainsi que les interfaces critiques comme les appels système. *Trinity* [48, 49] est un fuzzer pour les appels système de Linux. Il comporte dans sa base de connaissances un certain nombre d'annotations sur les appels système qui lui permettent de choisir de manière intelligente les arguments à passer aux appels système afin de déclencher des erreurs plus intéressantes que le rébarbatif code d'erreur `EINVAL` (paramètre invalide). En particulier, Trinity possède une notion de « type » d'objets Linux. Si un argument est censé être un descripteur de fichier, Trinity va tout d'abord ouvrir des fichiers de différents types (pseudo-fichiers, répertoires, tuyaux, sockets réseaux, etc.) pour en récupérer les descripteurs et les essayer, permettant ainsi de tester des chemins d'exécution au-delà du simple test de validité du descripteur. Trinity a permis la découverte de nombreux bugs [2], y compris hors de l'interface des appels système, dans la pile réseau notamment. Plus récent, *syzkaller* [22] emploie la même approche que Trinity en utilisant de l'information extraite du noyau pour fuzzer les appels système et est de plus capable d'examiner le code pour déterminer quels sont les chemins d'exécution qu'il a couverts. Cela lui permet de construire ses entrées de manière appropriée pour poursuivre le fuzzing. Cette méthode nécessite d'instrumenter le code du noyau à la compilation pour fournir des traces des chemins d'exécutions suivis. Le fuzzer syzkaller met également l'accent sur la reproductibilité des plantages, afin de construire des tests de régression [3].

2. Voir `http://codemonkey.org.uk/projects/trinity/bugs-found.php` pour une liste.

3. Un test de régression est un test effectué pour garantir qu'un problème absent d'une version n'apparaisse (ou ne réapparaisse) pas dans une version ultérieure.

De nombreuses instrumentations sont possibles avec les versions modernes du compilateur GCC. *KASAN* (*Kernel Address Sanitizer*) est une option de compilation du noyau Linux qui permet de produire une image de test du noyau où pour chaque groupe de huit octets, un octet est réservé pour tracer l'usage fait de la mémoire et en particulier surveiller qu'aucun objet alloué n'est utilisé avant d'être initialisé ni après avoir été libéré [25]. D'autres outils du même style existent comme *UBSAN (Undefined Behavior Sanitizer)* [78] qui surveille l'emploi d'opérations décrites comme ayant un comportement indéfini dans la sémantique du langage C tel que le déréférencement d'un pointeur nul, le dépassement de capacité d'un entier signé, etc. Ces outils sont particulièrement utiles en conjonction avec les fuzzers pour détecter les entrées qui poussent le noyau à adopter des comportements mal spécifiés.

D'autres projets et programmes existent pour améliorer le code du noyau Linux et en chasser les erreurs de programmation par des méthodes de test systématiques. Beyer et Petrenko militent par exemple pour une implémentation consolidée de tous les prototypes de l'état de l'art en matière de vérification logicielle, s'appuyant en particulier sur le *model checking*. Parmi les projets actifs portant sur le test du noyau Linux, et en particulier sur les pilotes de périphériques qui représentent une partie du code significative et très sujette aux erreurs à cause de l'interaction avec le matériel, on peut citer *Linux Driver Verification* [50], *Avinux* [71] et *DDVerify* [103].

2.5.4 Application au problème du positionnement des crochets LSM

Les travaux présents dans la littérature au sujet de la vérification du bon placement des crochets LSM sont dus à Jaeger et plusieurs équipes avec lesquelles il a collaboré. Nous donnons ici les grandes lignes de ces approches et nous comparerons leurs mérites respectifs dans le chapitre 5.

Vérification du bon positionnement des crochets

Les tous premiers travaux concernant la bonne position des crochets LSM ont été présentés en même temps que le framework lui-même en 2002 [112]. Il s'agit d'une vérification statique, portant sur le code C du noyau avec CQUAL, garantissant qu'un crochet LSM se trouve bien dans chaque chemin d'exécution avant une opération sensible. Les opérations sensibles sont identifiées comme étant des lectures ou modifications de certaines structures de données manipulées par les appels système. La vérification de sécurité est donc de « bas-niveau » dans le mesure où elle contrôle l'accès à des objets internes au noyau, et non à des abstractions de niveau utilisateur comme les fichiers, les sockets réseau, etc. Cette propriété seule n'est pas suffisante, quoiqu'essentielle. Dans un autre article [42], Jaeger, Edwards et Zhang proposent de vérifier que les bonnes autorisations sont réclamées pour chaque opération sensible. En effet, chaque crochet correspond à une fonction précise du module de sécurité et selon la fonction appelée, le module ne va pas fonder sa décision sur les mêmes éléments. Pour schématiser sur un exemple concret, si un processus donné a le droit de lire un certain fichier, mais pas de le modifier, il est vital que le crochet correspondant à l'autorisation de lecture soit appelé avant l'opération de lecture et le crochet correspondant au droit d'écriture avant l'opération d'écriture, sinon la mauvaise décision de sécurité est prise dans les deux cas.

Placement automatique des crochets

D'autres approches ont ensuite été présentées pour positionner automatiquement les crochets, ce qui présente l'avantage de faciliter la maintenance du noyau tout en continuant de maintenir les bonnes propriétés de sécurité désirées. Dans l'article « Leveraging "choice" to automate authorization hook placement » [63], Muthukumaran, Jaeger et Ganapathy présentent une méthodologie générale pour le placement correct et automatique de crochets de sécurité dans le cas général (et non limité à LSM). Ils valident ensuite leur approche en comparant avec le placement choisi par les développeurs experts des logiciels choisis pour exemple et exhibent quelques problèmes dans ceux-ci. Dans l'article « Maintaining the correctness of the Linux security modules framework » [26], Edwards et Jaeger considèrent uniquement LSM mais incluent l'aspect dynamique du noyau. En effet, le rythme de développement du noyau est rapide, avec une nouvelle version toutes les huit à dix semaines environ [4] et par conséquent, des vérifications de non-régression des propriétés de sécurité apportées par les crochets sont nécessaires.

À notre connaissance, il n'existe donc aucun travail ayant porté sur la position des crochets pour le suivi des flux d'information. En effet, LSM a été conçu premièrement pour implémenter des mécanismes de contrôle d'accès et n'a donc pas été prévu pour le contrôle de flux. Cependant, on peut constater qu'il existe d'ores et déjà des implémentations de suivi de flux d'information comme KBlare, Laminar et Weir s'appuyant sur LSM. La question de savoir si LSM constitue un framework viable pour l'implémentation de mécanisme de contrôle de flux est donc un problème ouvert et actuel, et consitue l'objet de cette thèse.

2.6 Conclusion

L'étude de la littérature dans le domaine montre une très grande diversité non seulement des outils de suivi de flux d'information mais aussi de leurs objectifs. Chaque auteur poursuit des garanties de confidentialité et d'intégrité différentes, et n'attribue pas toujours le même sens que les autres aux mêmes termes. Tandis que certains outils visent à couvrir le plus de canaux ouverts d'information possibles, d'autres visent à traiter le problème des canaux cachés. Certains outils encore se préoccupent de l'explosion des tags. On remarque cependant que la majorité de ces travaux se sont intéressés à des problématiques de politiques de flux, ou encore de modèle de programmation des applications. Notre revue de l'état de l'art a montré que le suivi de flux en soi, c'est-à-dire seulement la découverte des flux ayant lieu dans le système, a été semble-t-il négligé, ou du moins considéré comme un problème d'implémentation secondaire. On sait néanmoins qu'il n'en est rien car des recherches ont été conduites sur les problèmes de l'interposition dans les appels système et sur la bonne construction du framework LSM, bien que dans d'autres contextes que le suivi de flux d'information. Il reste, selon nous, un travail équivalent d'identification et de résolution des problèmes spécifique au suivi de flux. Autant que faire se peut, les méthodes formelles doivent être privilégiées. Même s'il est illusoire d'espérer aboutir à une implémentation « prouvée correcte » du suivi de flux dans le noyau Linux, les méthodes formelles donnent des garanties indispensables en termes de couverture du code et de maintenabilité dans le temps des implémentations des mécanismes de sécurité.

4. Le développement et les versions du noyau sont visibles sur `http://www.kernel.org`.

Chapitre 3

Conteneurs d'information du noyau Linux

Dans ce chapitre, nous décrivons quelques éléments de l'implémentation du noyau Linux afin de préciser l'objet de nos travaux de recherche et nos contributions des chapitres suivants. Nous donnons en premier quelques généralités pour situer le contexte de développement du noyau puis nous présentons les structures de données qui correspondent aux conteneurs d'information que nous avons vaguement introduit jusqu'alors. Ce chapitre a été rédigé en bonne partie sur la base des informations des ouvrages *Understanding the Linux Kernel* de Bovet et Cesati [6] et *Linux Kernel Development* de Love [54], ainsi que grâce au travail de Corbet, Edge et Sobol, éditeurs du site *LWN.net News from the source* [15].

3.1 Généralités

Linux est à l'origine le projet personnel d'un étudiant, Linus Torvalds, qui a commencé à le développer en 1991 après avoir fait l'acquisition d'un ordinateur muni d'un processeur Intel 80386. Le système est influencé par MINIX, qui était l'objet d'étude de Torvalds, mais la conception en est radicalement différente. En réalité, Linux est beaucoup plus classique et moins novateur du point de vue de l'architecture que nombre de systèmes d'exploitation plus anciens mais il met l'accent sur les performances et la stabilité.

Les sources du noyau Linux comportent, pour le version 4.7, environ quinze millions de lignes de code réparties sur près de quarante-trois mille fichiers [1] organisées en vingt-deux répertoires, présentés en table 3.1. Le développement du noyau est ouvert à tous, les sources sont sous licence libre (*Gnu Public License v2*) mais le processus d'intégration du code dans la version officielle du noyau est plutôt strict. Ceci garantit au noyau un style relativement homogène et une grande qualité malgré sa taille. Certains *bugs* peuvent cependant exister longtemps avant d'être découverts, du fait de la complexité du code [11].

Un accent particulier est mis sur les performances du noyau, sa stabilité ainsi que sur sa portabilité. En revanche, la sécurité ne reçoit pas d'attention particulière et est considérée comme un objectif ni plus ni moins important que les autres aux yeux de

1. Statistiques établies en utilisant `cloc` [19].

Torvalds [94]. Le noyau s'adapte à une grande variété de plate-formes et d'usages différents. On dénombre ainsi dans le dossier *arch* du code spécifique à vingt-huit architectures différentes, plus l'architecture spéciale *um* (pour *user-mode*) qui permet de lancer le noyau Linux comme un processus ordinaire, principalement pour le test ou le débogage. Le noyau Linux connaît un succès important sur les super-calculateurs [2], les serveurs web [3] et les téléphones portables et tablettes, à travers le système d'exploitation Android [4]. Il est donc tout aussi adapté à des environnements réclamant des performances extrêmes qu'à des environnements où la taille et la consommation énergétique du noyau doivent être minimales. Le noyau Linux peut également être utilisé en mode « temps réel », c'est-à-dire dans des environnements où il est crucial de borner le temps maximal de chaque action du noyau. Les logiciels manipulant des flux audios ou vidéos ou ceux dont dépendent la sécurité de personnes utilisent le mode temps réel.

3.2 Correspondances entre abstractions du système d'exploitation et structures de données internes du noyau

3.2.1 Système de fichiers virtuel

Le système de fichiers virtuel est une interface servant à unifier une grande partie du code des véritables systèmes de fichier et simplifier l'écriture de ces derniers. Il s'agit d'un ensemble de structures et de fonctions communes à tous les systèmes de fichiers. Le système de fichiers virtuel comprend également la définition des appels système liés à la manipulation des fichiers, répertoires, etc. On peut distinguer quatre structures de données principales :

struct super_block Un superbloc représente un système de fichiers monté dans l'arborescence des fichiers.

struct dentry Le terme *dentry* est l'abbréviation de *directory entry*, c'est-à-dire « entrée de répertoire ». Une entrée de répertoire est un nom de fichier accessible dans l'arborescence des fichiers. C'est un composant d'un chemin d'accès à un fichier.

struct inode Un *inode*, pour *index node*, généralement traduit en « i-nœud », est un fichier du système de fichiers, avec son contenu (ou plus exactement la position de son contenu sur le disque ou en mémoire) et ses méta-données mais pas son nom. En effet, un même i-nœud peut apparaître à plusieurs endroits dans le système de fichiers (et donc plusieurs entrées de répertoire peuvent y faire référence) ou même à aucun (le fichier est dans ce cas dit « anonyme »).

struct file Cette structure représente un descripteur de fichier, c'est-à-dire une référence sur un fichier ouvert par un processus. Posséder un descripteur de fichier est nécessaire pour la plupart des accès au contenu du fichier comme la lecture, l'écriture, la troncature, la projection en mémoire, etc.

Nous décrivons ci-après chaque structure de données.

2. En novembre 2016, Linux était le système d'exploitation de quatre cent quatre-vingt-dix-huit des cinq cents plus gros super-calculateurs [91].

3. 37,2% des parts de marché en mars 2017 [73]

4. 68,4% des parts de marchés en novembre 2016 [90].

Table 3.1 – Organisation des sources du noyau Linux

Répertoire	Documentation des API et fonctionnalités du noyau.
`arch`	Code variant selon l'architecture pour laquelle le noyau est compilé.
`block`	Code relatif aux périphériques en mode bloc.
`certs`	Code relatif aux certificats embarqués dans le noyau et à la signature du noyau et des modules.
`crypto`	Code des algorithmes et primitives cryptographiques (les implémentations dans le noyaux peuvent bénéficier d'accélération matérielle).
`drivers`	Code des pilotes de périphériques (c'est le répertoire le plus imposant du noyau, trois fois plus gros que `arch` et un à deux ordres de grandeur plus gros que les autres).
`firmware`	Images de *firmware* pouvant être intégrées dans les sources du noyau (compatible du point de vue de la licence).
`fs`	Code du système de fichiers virtuel ainsi que des systèmes de fichiers implémentés par Linux (chacun dans son sous-répertoire).
`include`	Fichiers d'en-tête du noyau, le sous-répertoire *linux* contient les entêtes utiles aux développeurs de bibliothèques et autres outils dépendant du noyau.
`init`	Code de chargement et initialisation du noyau commun à toutes les architectures.
`ipc`	Code des IPCs System V ainsi que des files de message POSIX.
`kernel`	Répertoire par défaut du code du noyau, contient notamment l'ordonnanceur, la définition et manipulation des *threads*, les horloges, les moyens de synchronisation, le code implémentant LSM, etc.
`lib`	Mini-bibliothèque pas tout à fait standard utilisable par le noyau (qui ne peut pas utiliser une implémentation tierce).
`mm`	Code relatif à la gestion de la mémoire, en particulier de la pagination.
`net`	Code des fonctions de réseau, et des protocoles associés (chacun dans son répertoire).
`samples`	Exemples d'usage d'API du noyau, à des fins de documentation ou de démonstration.
`scripts`	Code ne faisant pas partie des sources du noyau mais utile pour le développement, pour manipuler des *patchs* ou pour exporter des données utiles du noyau.
`security`	Code des modules implémentés avec LSM ainsi que du répertoire de clés du noyau.
`sound`	Code relatif à l'enregistrement, la reproduction, la manipulation du son.
`tools`	Code ne faisant pas partie du code du noyau mais implémentant des outils utilisables pour interroger le noyau, mettre en place des tests, etc.
`usr`	Code de génération de l'*initramfs*.
`virt`	Code relatif à la virtualisation (pour se servir de Linux comme hyperviseur ou comme machine virtuelle).

Systèmes de fichiers

La structure `struct super_block` représente un système de fichiers monté dans l'arborescence des fichiers. Elle contient notamment la liste de tous les i-nœuds du système de fichiers ainsi qu'un lien vers la racine du système. Il est possible pour un module LSM d'attacher une structure de sécurité à un système de fichiers, en particulier pour contrôler les opérations de montage, démontage, remontage et leurs options. Cependant, aucun des moniteurs de flux d'information étudiés ne considèrent les systèmes de fichiers comme des conteneurs d'information.

```
struct super_block {
    [...]
  unsigned long s_blocksize; /* Taille d'un bloc sur le disque */
  loff_t s_maxbytes; /* Taille maximale d'un fichier */
  struct file_system_type *s_type; /* Type du système de fichiers */
  const struct super_operations *s_op; /* Operations du système de
    fichiers : allocation d'un i-nœud, synchronisation sur le disque */
    [...]
  unsigned long s_magic; /* Identifiant du type de système de fichiers */
  struct dentry *s_root; /* Entrée de répertoire correspondant à la racine du
    système de fichiers */
    [...]
#ifdef CONFIG_SECURITY
  void *s_security; /* Structure de sécurité LSM */
#endif
  const struct xattr_handler **s_xattr; /* Opérations de manipulation
    des attributs étendus */
    [...]
  char s_id[32]; /* Nom du périphérique portant le système de fichiers */
  u8 s_uuid[16]; /* Identifiant unique du système de fichiers */
    [...]
  const struct dentry_operations *s_d_op; /* Opérations par défaut des
    entrées de répertoire du système de fichiers */
    [...]
  struct list_head s_inodes; /* Liste des i-nœuds du système de fichiers */
};
```

Entrées de répertoire

Les entrées de répertoire et leur organisation sous forme hiérarchique forment l'arborescence des fichiers dont les utilisateurs font l'expérience usuelle. Néanmoins, ce sont uniquement des noms, des liens, permettant d'accéder aux fichiers et non une représentation de ceux-ci.

Les entrées de répertoire ne possèdent pas de structure LSM car il ne correspondent pas à une abstraction présentée aux processus utilisateur. En effet, les processus manipulent essentiellement les fichiers sous forme d'i-nœuds ou bien via un descripteur de fichier. Lorsqu'un appel système prend un nom de fichier ou un chemin en paramètre, c'est sous la forme d'une chaîne de caractère. Les structures de type `struct dentry` représentant les entrées de répertoire sont en fait créées à la demande, lorsqu'un chemin est donné sous la forme d'une chaîne de caractères et que le fichier correspondant doit être recherché dans l'arborescence des fichiers. Si un module LSM, ou même un

moniteur de flux d'information, souhaite contrôler la création, la suppression ou le renommage d'entrées de répertoire, il peut le faire en fonction de la structure associée à l'i-nœud du répertoire et du chemin. En effet, renommer un fichier consiste en la modification de l'inode représentant le dossier contenant ce fichier. Pour le noyau, un dossier est un fichier dont le contenu est la liste des fichiers et sous-dossiers rangés à l'intérieur.

```
struct dentry {
    [...]
   struct dentry *d_parent; /* Répertoire parent */
   struct qstr d_name; /* Nom de l'entrée de répertoire */
   struct inode *d_inode; /* I-nœud de l'entrée de répertoire */
    [...]
   const struct dentry_operations *d_op; /* Opérations de l'entrée de
     répertoire */
   struct super_block *d_sb; /* Le système de fichiers de l'entrée de répertoire */
   void *d_fsdata; /* Données de l'entrée de répertoire spécifiques au système de
     fichiers */
    [...]
   struct list_head d_child; /* Liste des sous-répertoires du répertoire parent */
   struct list_head d_subdirs; /* Liste des sous-répertoires de la structure
     courante */
    [...]
};
```

I-nœuds

L'i-nœud est la structure centrale du système de fichiers virtuel. Un i-nœud représente ce qui apparaît communément aux yeux des processus utilisateurs comme un fichier, c'est-à-dire une suite d'octets qui peut être lue et éventuellement écrite selon le type de fichier. Les i-nœuds représentent la partie commune à tous les types de fichiers : fichiers réguliers, répertoire, tubes, sockets réseau, périphériques, fichiers spéciaux qui, à la lecture, renvoient la sortie d'une fonction du noyau, etc. Les champs particuliers associés à certains types de fichiers sont rangés dans une structure à part, accessible depuis l'i-nœud. L'i-nœud représente à la fois le contenu du fichier et ses méta-données comme sa taille, la dernière fois qu'il a été modifié, etc. Cette structure appartenant au système de fichiers virtuel, un champ est prévu pour que le vrai système de fichiers portant l'i-nœud y associe ses données particulières, entre autres la localisation du contenu du fichier. Les i-nœuds sont stockés de manière permanente sur le disque. Les instances de la structure `struct inode` sont créées dès que le fichier est utilisé, et peuplées depuis les informations sur le disque. La correspondance entre les i-nœuds et les entrées de répertoire est laissée libre à chaque système de fichiers.

Cette structure possède une structure LSM qui est souvent, selon les capacités du système de fichiers, sérialisée avec le contenu de l'i-nœud et qui est donc persistante d'un redémarrage à l'autre du système. C'est la principale structure sur laquelle s'appuient les modules LSM pour prendre leurs décisions de sécurité.

```
struct inode {
   umode_t i_mode; /* Mode et permissions UNIX de l'i-nœud */
    [...]
   kuid_t i_uid; /* Identifiant du propriétaire */
```

```
  kgid_t i_gid; /* Identifiant du groupe propriétaire */
   [...]
#ifdef CONFIG_FS_POSIX_ACL
  struct posix_acl *i_acl; /* Permission étendues */
   [...]
#endif
  const struct inode_operations *i_op; /* Opérations de l'i-nœud */
  struct super_block *i_sb; /* Système de fichiers auquel appartient l'i-nœud */
  struct address_space *i_mapping; /* Liste des espaces d'adressage de
    processus dans lequel le fichier est projeté */
#ifdef CONFIG_SECURITY
  void *i_security; /* Structure de sécurité LSM */
#endif
  unsigned long i_ino; /* Identifiant interne de l'i-nœud */
   [...]
  const unsigned int i_nlink; /* Nombre d'entrées de répertoires associées à
    l'i-nœud */
   [...]
  loff_t i_size; /* Taille de l'i-nœud */
  struct timespec i_atime; /* Dernier horodatage d'accès */
  struct timespec i_mtime; /* Dernier horodatage de modification */
  struct timespec i_ctime; /* Horodatage de création */
   [...]
  const struct file_operations *i_fop; /* Opérations par défaut des
    descripteurs de fichier de cet i-nœud */
   [...]
  union { /* Structure de données particulière dépendant du type de fichier */
     struct pipe_inode_info *i_pipe; /* Structure de données propre aux
    tubes */
     struct block_device *i_bdev; /* Structure de données propre aux
    périphériques en mode bloc */
     struct cdev *i_cdev; /* Structure de données propre aux périphériques en
    mode caractère */
     char *i_link; /* Donnée propre aux liens symboliques (en fait, c'est le chemin du
    fichier pointé) */
     unsigned i_dir_seq; /* Donnée propre aux répertoires, servant à synchroniser
    les accès à la structure. */
  };
   [...]
  void *i_private; /* Données de l'i-nœud spécifique au système de fichiers */
};
```

Descripteurs de fichier

Chaque *thread* peut possèder sa propre table de descripteurs de fichiers ou bien partager celle de son *thread* parent. Cette table référence les fichiers qu'il a ouverts. Un descripteur de fichier est obtenu par l'appel système `open`. En plus de l'entrée de répertoire passée à l'ouverture, le descripteur de fichier possède un curseur, indiquant où sera effectué la prochaine lecture ou écriture dans le fichier (sauf cas particuliers des fichiers séquentiels, qui ne peuvent être lus que dans un ordre strict, et des fichiers ouverts en mode `O_APPEND`, où toute écriture s'effectue à la fin).

Il existe une structure de sécurité LSM dans la structure `struct file` mais elle est peu utilisée par les moniteurs de flux d'information car représenter le contenu du fichier, et donc ses teintes, est de la responsabilité de l'i-nœud du fichier, et non de ses descripteurs. En effet, si un processus modifie le fichier via son descripteur, tous les autres processus voient les modifications même s'ils ne partagent pas le même descripteur de fichiers car les modifications portent sur l'i-nœud référencé par le descripteur et non le descripteur lui-même.

```
struct file {
   [...]
  struct path f_path; /* Chemin du fichier représenté par le descripteur */
  struct inode *f_inode; /* I-nœud du descripteur de fichier (en cache) */
  const struct file_operations *f_op;/* Opérations du descripteur de
    fichier (lecture, écriture, déplacement, etc.) */
   [...]
  fmode_t f_mode; /* Mode et permission d'ouverture du fichier */
   [...]
  loff_t f_pos; /* Position de lecture/écriture dans le fichier */
   [...]
  const struct cred *f_cred; /* Autorisations du processus ayant ouvert ce
    descripteur de fichiers */
   [...]
#ifdef CONFIG_SECURITY
  void *f_security; /* Structure de sécurité LSM */
#endif
  void *private_data; /* Données associées au descripteur dépendant du fichier
    sous-jacent */
   [...]
  struct address_space *f_mapping; /* Espaces d'adressage de processus
    dans lequel le fichier est projeté */
   [...]
};
```

3.2.2 Structures de données relatives aux processus

Processus et *threads*

Un processus est un objet du système d'exploitation dont l'activité consiste à charger un programme écrit dans un fichier et à l'exécuter. Son existence est déterminée par le début et la fin du programme exécuté. Un processus dispose de ressources de la part du système d'exploitation. Une des tâches essentielles du noyau est d'arbitrer l'usage des ressources entre les processus, à savoir le processeur, la mémoire, les périphériques, etc. C'est au noyau que revient la charge d'attribuer des quotas de temps pendant lesquels les processus peuvent effectivement s'exécuter. Lorsque ce quota est épuisé, ou bien lorsque le processus relâche volontairement le processeur — dans l'attente d'une entrée–sortie, typiquement — le noyau sauvegarde l'état du processus puis choisit un autre processus pouvant s'exécuter, restaure l'état de ce dernier (les registres du processeur essentiellement) et enfin relance son exécution. Pendant ce temps, le processus interrompu dans son exécution est mis en pause. Cette opération s'appelle le *changement de contexte*. La mission du noyau consistant à attribuer les quotas de temps

à chaque processus, à choisir le bon candidat pour s'exécuter et pour lancer ou interrompre les exécutions s'appellent l'*ordonnancement*. L'ordonnanceur est une portion du code du noyau particulièrement centrale et critique, et également techniquement complexe. De plus, elle dépend grandement de l'architecture du processeur.

Un processus peut être composé de plusieurs séquences d'exécution parallèles, appelées *threads*. Par exemple, un serveur web peut traiter plusieurs requêtes simultanément en déléguant le traitement de chacune à un *thread*. Les *threads* sont également connus sous le nom de processus légers car les *threads* d'un même processus partagent leur contexte d'exécution, au contraire des processus qui sont très isolés les uns des autres. En particulier, tous les *threads* d'un même processus partagent la même mémoire, les mêmes gestionnaires de signaux et en général, la même liste de descripteurs de fichiers ouverts.

Dans le noyau, il n'y a qu'une seule structure de données représentant l'ensemble des objets exécutant du code, les *tâches*, décrites par la structure `struct task_struct` [97, `include/linux/sched.h`]. C'est une structure particulièrement longue et dépendante des options de compilation. On n'en représente ici qu'une petite partie.

```
struct task_struct {
  volatile long state; /* Indicateur de l'activité de la tâche */
   [...]
  unsigned int ptrace; /*Indique si un traçage de la tâche est en cours */
   [...]
  int prio, static_prio, normal_prio; /*Information de priorité et de
    préférences d'ordonnancement */
  unsigned int rt_priority;
  const struct sched_class *sched_class;
   [...]
  cpumask_t cpus_allowed; /*Processeurs sur lesquels la tâche est autorisée à
    s'exécuter */
   [...]
  struct list_head tasks; /*Liste de toutes les tâches */
   [...]
  struct mm_struct *mm, *active_mm; /*Mémoire de la tâche, comme les
    tâches du noyau n'ont pas de mémoire propre, elles empruntent celles de la dernière tâche
    utilisateur à s'être exécutée, d'où la distinction entre mm et active_mm */
   [...]
  int exit_state;   /*État dans lequel la tâche s'est terminée, ces informations sont à
    disposition de la tâche parente après la mort d'une tâche */
  int exit_code, exit_signal;
  int pdeath_signal; /*Signal émis par une tâche à destination de ses tâches-filles
    lorsqu'elle meurt, optionnelle */
   [...]
  pid_t pid; /*Identifiant de la tâche */
  pid_t tgid; /*Identifiant de groupe de la tâche */
   [...]
  struct task_struct __rcu *real_parent; /*Tâche ayant réellement
    donnée naissance à la tâche courante */
  struct task_struct __rcu *parent; /*Tâche à laquelle la tâche courante
    doit signaler sa mort */
  struct list_head children; /*Liste des tâches-filles */
  struct list_head sibling; /*Entrée dans la liste de tâches-filles de la tâche
    parente */
  struct task_struct *group_leader; /*Tâche du même groupe que la tâche
```

```
    courante dont l'identifiant est l'identifiant de groupe, elle est désignée comme leader du
    groupe (il peut s'agir de la tâche courante elle-même) */
  struct list_head ptraced; /*Liste des tâches suivies avec ptrace par la tâche
    courante */
  struct list_head ptrace_entry; /*Entrée dans la liste des tâches suivies par
    la tâche parente, le cas échéant */
   [...]
  struct list_head thread_group; /*Liste des tâches du même groupe */
  struct list_head thread_node; /*Entrée dans la liste des tâches du même
    groupe */
   [...]
  cputime_t utime, stime, utimescaled, stimescaled; /*Divers
    compteurs pour évaluer le temps d'exécution de la tâche */
  cputime_t gtime;
   [...]
  unsigned long nvcsw, nivcsw; /*Compteurs du nombre de changements de
    contexte de la tâche */
  u64 start_time; /*Heure de démarrage de la tâche */
   [...]
  /* process credentials */
  const struct cred __rcu *real_cred; /*Autorisations réelles de la tâche
    (voir sous-section 3.2.2) */
  const struct cred __rcu *cred; /*Autorisations affichées de la tâche */
  char comm[TASK_COMM_LEN]; /*Ligne de commande ayant démarré la tâche */
#ifdef CONFIG_SYSVIPC
  struct sysv_sem sysvsem; /*Informations sur les sémaphores System V
    possédés */
  struct sysv_shm sysvshm; /*Informations sur les mémoires partagées System V
    possédés */
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
  unsigned long last_switch_count; /*Détecteur de tâches bloquées, c'est un
    compteur du nombre de fois que la tâche a été sélectionnée par l'ordonnanceur pour
    s'exécuter */
#endif
  struct fs_struct *fs; /*Informations sur le système de fichiers, notamment le
    répertoire de travail courant */
  struct files_struct *files; /*Liste des descripteurs de fichiers ouverts */
  struct nsproxy *nsproxy; /*Espaces de nommage dont la tâche fait partie */
  struct signal_struct *signal; /*État des signaux */
  struct sighand_struct *sighand; /*Gestionnaires de signaux installés */
  sigset_t blocked, real_blocked; /*Ensembles des signaux bloqués */
  sigset_t saved_sigmask; /*Masque des signaux sauvé */
  struct sigpending pending; /*Signaux reçus et en attente de traitement */
   [...]
  struct thread_struct thread; /*État de la tâche en cours sauvé chaque fois
    que la tâche est stoppée par le noyau. Cet état est nécessaire pour remettre le CPU dans
    l'état précis où la tâche a été interrompue la prochaine fois qu'elle est sélectionnée par
    l'ordonnanceur, cette structure est définie de façon différente selon l'architecture. */
};
```

L'ordonnanceur du noyau ne connaît que les tâches, qui correspondent plutôt aux *threads* qu'aux processus. Pour ajouter à la confusion, le code du noyau utilise tantôt

TABLE 3.2 – Ressources pouvant être partagées entre une tâche et la nouvelle tâche qu'elle crée

Élément partageable	Description
Espace d'adressage	Partage de toutes les zones mémoires, à l'exception de la pile qui est toujours traitée à part et locale à chaque tâche
Système de fichiers	Partage des informations à propos du système de fichiers, par exemple, la racine et le répertoire courant
Descripteurs de fichiers	Partage des descripteurs de fichiers ouverts
Gestionnaires de signaux	Partage de la liste des gestionnaires de signaux installés
Ptraçage	Si la tâche parente est ptracée, la nouvelle tâche l'est aussi
Parent	Partage de la tâche parente ; en d'autres termes, la tâche créant la nouvelle tâche ne devient pas la parente de cette dernière mais sa sœur
Thread	Partage du même groupe de tâches, la nouvelle tâche créée correspond donc à un nouveau *thread* du même processus que la tâche courante
Sémaphores System V	Partage des sémaphores System V (servant à la synchronisation entre tâches)
Contexte d'entrée–sortie	Partage des mêmes informations permettant de manipuler les périphériques via leurs registres ou autres méthodes d'accès direct
Espace de nommage « Système de fichiers »	Partage du même système de fichiers virtuel (même racine, même arborescence de fichiers visible)
Espace de nommage « CGroup »	Partage des mêmes limitations et quotas sur les ressources (comme les processeurs, la mémoire, etc.)
Espace de nommage « UTS »	Partage des informations sur le nom de la inter-machine et autres identifications réseau
Espace de nommage « IPC »	Partage des informations sur les IPC System 5 ; une IPC n'est accessible via sa clé qu'à l'intérieur de l'espace de nommage où elle a été créée
Espace de nommage « Utilisateurs »	Partage des utilisateurs enregistrés dans le système
Espace de nommage « PID »	Partage de la liste des tâches en cours d'exécution
Espace de nommage « Réseau »	Partage des interfaces réseau, de la tâble de routage, etc.

Les espaces de nommages sont partagés par défaut. Les autres éléments ne sont pas partagés si l'appel système `fork` est utilisé. Seul l'appel système `clone`, plus récent, permet de contrôler finement les éléments partagés.

le mot « *thread* » (comme dans `threads_struct`), tantôt le mot « processus » (comme dans `pid` qui est l'abbréviation de *Process IDentifier*) pour désigner en réalité une tâche. Cela est dû au fait qu'à l'origine, le noyau ne connaissait que les processus et aucun support n'était prévu dans le noyau pour les *threads*. Ces derniers étaient implémentés en espace utilisateur. L'introduction de l'ordonnancement des *threads* dans la version 2.0 du noyau a permis un gain en termes de performances, au prix d'une complexité accrue du code, et d'une certaine confusion chez les programmeurs système débutants tentant de comprendre le noyau.

Les tâches peuvent être groupées : les tâches partageant le même champ `tgid` (*thread group identifier*) font partie du même groupe. L'identifiant `tgid` est égal à l'identifiant de la tâche leader du groupe, en général la première tâche de ce groupe. Conceptuellement, un groupe de tâche correspond à peu près à un processus. Dans les faits, Linux généralise les concepts de processus et de *thread* en permettant à deux tâches de partager une certaine partie de leurs ressources (présentées en table 3.2) tout en en conservant d'autres strictement séparées. Une tâche peut par exemple partager avec une autre sa table de descripteurs de fichiers, ou bien sa mémoire, ou encore ses sémaphores, sans partager le reste. Par la suite, l'appel système `unshare` permet de *dé-partager* cet état commun, en dupliquant les structures concernées. Ce modèle est donc plus riche que la dichotomie *thread*/processus mais il est rare que les programmes utilisateur en profitent pleinement. En revanche, certaines bibliothèques majeures utilisent ce mécanisme. Il existe cependant quelques restrictions sur les ressources partageables. En effet, le modèle UNIX et les normes qui l'ont suivies imposent que lorsqu'un signal est envoyé à un processus, tous les *threads* puissent le recevoir. Par conséquent, le noyau fait en sorte que lorsque deux tâches partagent le même identifiant de groupe (c'est-à-dire qu'elles représentent deux *threads* du même processus), elles partagent leur gestionnaire de signaux, ce qui implique également le partage de leur mémoire.

Autorisations

Les autorisations sont des informations associées à un objet et comprenant [97, file `Documents/security/credentials.txt`] :

- les informations sur son propriétaire et son groupe propriétaire ;
- les droits d'accès qui lui sont associés ;
- les permissions qui lui sont accordées pour agir sur d'autres objets du système ;
- un état de sécurité nécessaire à un module LSM, le cas échéant.

Le terme « autorisations » est une traduction imparfaite de l'anglais *credentials* qui évoque les qualifications ou les diplômes d'une personne qui se présenterait à un emploi ; de la même manière qu'un processus doit présenter des permissions suffisantes au noyau pour ouvrir un fichier. La structure `struct cred`, associée aux processus, est définie dans `include/linux/cred.h`. Tous les *threads* d'un processus partagent la même structure d'autorisations. Les autres objets du système comme les fichiers ont une structure d'autorisations moins grande car ils n'ont pas par exemple de *capacités*. En revanche, les fichiers exécutables possèdent une caractéristique unique qui est le couple de bits `SUID`/`SGID` qui confère au processus exécutant ce fichier les permissions du propriétaire/groupe propriétaire du fichier.

```
struct cred {
   [...]
  kuid_t uid; /* Utilisateur propriétaire réel */
  kgid_t gid; /* Groupe propriétaire réel */
  kuid_t suid; /* Utilisateur propriétaire sauvegardé */
  kgid_t sgid; /* Groupe propriétaire sauvegardé */
  kuid_t euid; /* Utilisateur propriétaire affiché */
  kgid_t egid; /* Groupe propriétaire affiché */
  kuid_t fsuid; /* Utilisateur utilisé pour les opérations sur les fichiers */
  kgid_t fsgid; /* Groupe utilisé pour les opérations sur les fichiers */
   [...]
  kernel_cap_t cap_inheritable; /* Capacités dont les processus-fils héritent */
  kernel_cap_t cap_permitted; /* Capacités pouvant être possédées */
  kernel_cap_t cap_effective; /* Capacités dont il peut être fait usage */
  kernel_cap_t cap_bset; /* Capacités qu'il est permis de transmettre par
    héritage */
   [...]
#ifdef CONFIG_SECURITY
  void *security; /* Structure de sécurité LSM */
#endif
  struct user_struct *user; /* Utilisateur réel responsable (pour la gestion des
    quotas notamment) */
  struct user_namespace *user_ns; /* Espace de nommage dans lequel cette
    structure est utilisée */
  struct group_info *group_info; /* Groupes supplémentaires déterminant les
    permissions sur les fichiers */
   [...]
};
```

3.2.3 Structures de données relatives à la mémoire

La mémoire de chaque processus constitue un conteneur d'information. En effet, c'est à l'intérieur de la mémoire qui lui est allouée qu'un processus peut récupérer le contenu d'un fichier lors d'une lecture, ou préparer un message à envoyer via une file de messages par exemple. De plus, la mémoire sert de tampon à toutes les opérations d'entrées–sorties vers les fichiers et les périphériques matériels. Dans cette section, nous décrivons comment la mémoire est divisée en pages, organisée et attribuée aux processus.

Cache des pages

Lorsqu'un processus lit ou écrit dans un fichier stocké sur un disque, cela ne se traduit pas systématiquement en une entrée–sortie vers le périphérique physique. En effet, une requête vers ledit périphérique prend un temps relativement long, en comparaison d'un accès à la mémoire, en raison du matériel lui-même. Lire depuis un disque dur magnétique requiert notamment d'écrire dans un certain registre du micro-contrôleur du disque dur, qui doit ensuite interpréter cette requête, déplacer la tête de lecture des disques, effectuer la lecture pour rapatrier les données dans une petite mémoire interne puis enfin copier les données dans la mémoire centrale pour que le processeur puisse les exploiter. Il y a donc en général un mécanisme de

cache : une portion de la mémoire est dédiée à la représentation du contenu des fichiers. Lorsqu'un fichier est lu pour la première fois, une requête normale est effectuée à destination du disque et une partie du contenu du fichier est rapatriée dans le cache. Les lectures et écriture successives se font directement depuis le cache, qui agit comme un *proxy* du disque. Si une lecture porte sur une partie du fichier absente du cache, une nouvelle requête vers le disque est faite pour le peupler. Naturellement, le cache a une taille limitée et parfois, il est nécessaire d'évincer certains fichiers. Le noyau choisit soigneusement les fichiers à éliminer en fonction de la dernière fois qu'ils ont été utilisés et de la probabilité qu'ils le soient à nouveau dans un futur proche. Tout comme le reste de la mémoire, le cache est divisé en pages d'une certaine taille (la sous-section suivante décrit le concept de pagination plus en détail). Chaque page est manipulée indépendamment et possède un statut permettant de tracer son usage. En particulier, une page est déclarée « sale » si elle a été écrite dans le cache mais que les modifications n'ont pas encore été répercutées sur le support physique du fichier, le disque. C'est un *thread* du noyau qui, de manière asynchrone, recopie le cache sur les disques, en veillant à ne pas impacter les performances du système tout en essayant de protéger les données des utilisateurs. En effet, laisser trop de pages sales expose les utilisateurs au risque de perdre beaucoup de données en cas d'arrêt inopiné de la machine. L'utilisation du cache présente également l'intérêt d'unifier une grande partie du code des systèmes de fichiers. Les supports physiques des systèmes de fichiers peuvent être variés : disque dur, stockage sur le réseau, mémoire (pour les systèmes de fichiers temporaires), etc. mais presque tous peuvent bénéficier du cache, une exception notable étant les pseudo-systèmes de fichiers pour lesquels la lecture d'un fichier est en fait la sortie d'une fonction du noyau. La gestion du cache est donc commune et les systèmes de fichiers ont seulement à se préoccuper d'organiser le stockage et de faire l'interface entre le pilote du périphérique de stockage et le système de fichiers virtuel. Le cache n'est cependant pas bénéfique pour tous les cas d'usage et il est possible pour les applications qui le souhaitent de le court-circuiter.

Mémoire des processus

Les processus sont isolés les uns des autres. Ils ne peuvent accéder directement qu'à leur propre mémoire. Le noyau applique cette isolation en introduisant un niveau d'indirection dans la façon dont les adresses mémoire utilisées par les processus sont interprétées : la *pagination*. La mémoire physique de la machine est découpée en un ensemble de pages d'une certaine taille, habituellement $4\,\mathrm{Kio}$. Si une adresse est écrite sur 32 bits et référence un octet en particulier, elle est donc composée de 20 bits identifiant la page plus 12 bits ($2^{12}\,\mathrm{o} = 4096\,\mathrm{o} = 4\,\mathrm{Kio}$) donnant le décalage, à l'intérieur de cette page, de l'octet identifié [5]. Il existe deux sortes d'adresses. Les adresses physiques font référence à la véritable mémoire de la machine : seul le noyau peut s'en servir. Les adresses virtuelles sont quant à elles converties par le processeur et le noyau en adresses physiques de manière différente selon le processus qui s'exécute. Cette traduction est rapide car en général elle est faite par un composant matériel du processeur, que le noyau programme à l'avance. Cela est réalisé en maintenant une correspondance entre les pages virtuelles de chaque processus et les pages physiques. Les bits indiquant le décalage à l'intérieur de la page restent, eux, identiques. Cette indirection a deux avantages. D'une part, plusieurs processus peuvent utiliser les mêmes adresses virtuelles sans avoir à se coordonner. En fait, ils peuvent même utiliser

5. Du moins, c'est conceptuellement ce qui se passe ; l'implémentation peut être plus ou moins sordide, selon l'architecture considérée.

plus de mémoire virtuelle qu'il n'y a de mémoire physique dans la machine : les pages manquantes sont simplement prises sur le disque dur plutôt que dans la mémoire centrale. Ensuite, la traduction des adresses virtuelles en adresses physiques donne l'opportunité de détecter des erreurs de manipulation ou des attaques délibérées des processus. Par exemple, si un processus essaie d'utiliser une adresse invalide, il est facile de le repérer car sa traduction en adresse physique ne correspondra à aucune page autorisée.

La mémoire de chaque processus, également appelée *espace d'adressage* car il s'agit de l'ensemble des adresses que peut utiliser le processus, est décrite par une structure `struct mm_struct`. Le champ le plus intéressant de cette structure est `struct mm_struct *mmap` qui est une liste chaînée des zones de mémoire attribuées au processus. Une zone de mémoire correspond à un intervalle d'adresses virtuelles qu'un processus est autorisé à utiliser. Par exemple, la section de code de son programme correspond à une zone de mémoire d'un processus. Si deux processus partagent des pages de mémoire, la section de mémoire partagée compte dans la mémoire de chacun des processus comme une zone de mémoire (et grâce au mécanisme de pagination, elle pourra même ne pas débuter à la même adresse virtuelle dans les deux processus). Les permissions d'accès sont attribuées zone par zone. Toutes les zones sont lisibles ; en revanche, seules les pages de code sont ordinairement exécutables et beaucoup de pages ne sont pas modifiables.

La présence d'un cache des fichiers ouverts permet également la mise en place d'un mécanisme supplémentaire de lecture–écriture dans les fichiers : la *projection en mémoire*. La projection d'un fichier consiste à réserver une zone de mémoire dans l'espace d'adressage d'un processus de telle sorte que les adresses virtuelles de cette zone soient traduites en des adresses des pages de ce fichier dans le cache. En d'autres termes, les accès à la mémoire par le processus projecteur sont traduits en des accès directs au cache, ce qui équivaut à lire ou écrire le fichier, sans faire d'appels système de la famille de `read` ou `write`. Il y a plusieurs sortes de projection. Les projections *partagées* sont celles utilisant réellement les pages de cache à la fois pour les lire et les modifier, de sorte que tous les autres processus du système verront les modifications, comme si elles avaient été faites en écrivant dans le fichier avec un appel système de type `write`. Le deuxième type de projection sont les projections *privées*. Elles portent elles aussi sur les pages de cache, mais uniquement *jusqu'à la première écriture*. Lors de l'écriture, les pages sont copiées dans la mémoire du processus, qui lit et modifie donc la copie, et non l'original. Les modifications qu'il apporte ne sont pas visibles par les autres processus, et il ne voit pas les modifications des autres processus dans sa projection. Loin d'être une fonctionnalité obscure du système, la projection de fichiers en mémoire est omniprésente sous Linux. En effet, c'est de cette manière que les processus exécutent leur programme. Un exécutable est partagé en plusieurs sections : le code, les données statiques, les constantes, etc. Chacune de ces sections est projetée dans la mémoire du processus lorsqu'il exécute un programme. Les bibliothèques que le programme requiert sont elles aussi traitées de la même manière. Cela permet à une bibliothèque très souvent utilisée, comme la bibliothèque standard du C, d'être chargée une seule fois en mémoire physique, et que chaque processus puisse la voir dans son propre espace d'adressage. La projection de fichiers sert également à mettre en place des sections de mémoire partagée entre plusieurs processus. La section 3.2.4 détaille ce mécanisme.

La structure `vm_area_struct` est définie comme suit [97, `include/linux/mm_types.h`].

```
struct vm_area_struct {
  unsigned long vm_start; /*Adresse virtuelle de début de la zone */
  unsigned long vm_end; /*Adresse virtuelle de fin de la zone */
  struct vm_area_struct *vm_next, *vm_prev; /*Liste chaînée de toutes
    les zones de mémoire du processus */
   [...]
  struct mm_struct *vm_mm; /*Espace d'adressage à laquelle appartient la zone de
    mémoire */
  pgprot_t vm_page_prot; /*Droits d'accès à la zone de mémoire */
   [...]
  struct list_head anon_vma_chain; /*Liste des zones de mémoire anonymes
    (c'est-à-dire n'étant pas des projections effectuées depuis des fichiers) dans le même
    espace d'adressage */
  const struct vm_operations_struct *vm_ops; /*Opérations sur la zone
    de mémoire */
  unsigned long vm_pgoff; /*Décalage dans le fichier de la projection de la zone
    de mémoire */
  struct file * vm_file; /*Fichier projeté dans la zone */
   [...]
};
```

3.2.4 Structures correspondant aux canaux de communication entre processus

Contrairement aux fichiers qui représentent un moyen de stockage pérenne de l'information, les canaux d'Inter-Process Communication (IPC)* sont utilisés pour la communication entre les processus. Ils n'ont pas vocation à persister dans le temps, ni à stocker de l'information, mais seulement à l'acheminer. Nous distinguons quatre IPC différents :

- les tubes ;
- les files de messages ;
- les mémoires partagées ;
- les *sockets* réseau.

Nous ne considérons pas ici, comme il est usuel de le faire, les sémaphores dans la liste des IPC car ils ont une fonction de synchronisation et non d'acheminement de l'information. Néanmoins, ils forment un canal caché de stockage parfaitement fiable et d'une bande passante non négligeable.

Pour des raisons historiques, certaines IPC sont en double dans le noyau Linux. Ainsi, il existe une version des files de messages et des mémoires partagées héritées de System V, le système propriétaire UNIX développé par l'entreprise AT&T, et une autre plus récente, conforme à la description donnée dans la norme Portable Operating System Interface (POSIX)*. Afin de maintenir la compatibilité avec d'anciens programmes, le noyau doit continuer à supporter les deux interfaces côté processus utilisateur. Cependant, côté noyau, le code des deux API de files de messages et des deux API de mémoire partagée, respectivement, sont unifiés.

Tubes

Les tubes, plus connus sous le nom de *pipes* en anglais, sont des ensembles de tampons de mémoire du noyau se présentant du point de vue des processus utilisateurs comme des fichiers un peu particuliers. Les tubes ne fonctionnent que si (au moins) deux processus sont connectés : un écrivain et un lecteur. Contrairement à un fichier normal, les données ne peuvent être lues que de manière strictement séquentielle : un octet écrit avant un autre est également lu avant. Les tubes ont une capacité maximale. Si le lecteur essaie de lire alors que le tube est vide, ou si l'écrivain essaie d'écrire dans le tube alors qu'il est plein, l'opération bloque et le processus est endormi dans l'attente que l'autre processus remplisse ou vide (respectivement) le tube. Il est également possible de se servir du tube en mode non-bloquant, dans ce cas, si une opération aurait dû bloquer en temps normal, elle n'effectue qu'une lecture ou écriture partielle et renvoie son résultat. Les tubes peuvent être nommés, ils possèdent alors une ou des entrées dans le système de fichiers. Ces tubes sont créés par l'appel système `mknod`. Ils peuvent également être anonymes et dans ce cas ils sont créés par l'appel système `pipe`. Cet appel système renvoie deux descripteurs de fichier, l'un pour lire depuis le tube et l'autre pour y écrire. L'usage habituel de ces descripteurs est que le processus faisant l'appel se clone ensuite, le processus-fils est alors créé avec une copie des descripeurs de fichiers et les deux processus peuvent exécuter chacun leur code et communiquer via le tube. Il est également possible pour un processus de faire parvenir un descripteur de fichier à un autre processus via une socket. En plus des champs communs à tous les i-nœuds, la structure suivante est attachée au sein de chaque i-nœud représentant un tube, dans le champ `i_pipe` (voir la structure `struct inode` page 39).

```
struct pipe_inode_info {
   [...]
  wait_queue_head_t wait; /* File d'attente en cas de tube plein ou vide */
  unsigned int buffers; /* Nombre de tampons individuels composant le tube */
  unsigned int nrbufs; /* Nombre de tampons non-vides dans le tube */
  unsigned int curbuf; /* Indice du tampon courant */
  unsigned int readers; /* Nombre de lecteurs du tube */
  unsigned int writers; /* Nombre d'écrivains du tube */
  unsigned int files; /* Nombre de descripteurs de fichiers référençant l'i-nœud
    du tube */
  unsigned int waiting_writers; /* Nombre d'écrivains bloqués dans le
    noyau */
   [...]
  struct pipe_buffer *bufs; /* Groupe des tampons composant le tube */
  struct user_struct *user; /* Utilisateur ayant créé le tube */
};
```

Mémoires partagées

Partager une portion de mémoire consiste à allouer dans deux processus ou plus une plage d'adresses qui correspond aux mêmes pages de mémoire physique. De la sorte, chaque processus utilisant une adresse de cette plage lit et écrit non seulement dans sa mémoire, mais aussi celle des autres processus automatiquement. Le mécanisme est le même que celui de la projection en mémoire de fichiers, à ceci près que le fichier n'est pas tenu d'exister réellement sur le disque, il peut s'agir d'un fichier temporaire

— existant uniquement en mémoire, sans avoir de contrepartie sur un disque — et anonyme — n'apparaissant pas dans l'arborescence des fichiers. Nous commençons par décrire dans cette section les mémoires partagées POSIX implémentées en termes de mécanismes déjà décrits dans ce chapitre. Bien qu'en réalité plus anciennes, les mémoires partagées System V ne font plus office aujourd'hui que de surcouche aux mémoires partagées POSIX.

Mettre en place une mémoire partagée consiste pour chacun des processus impliqués à :

1. ouvrir un même fichier ;
2. projeter ce fichier en mémoire de manière partagée.

S'il n'est pas souhaitable que le fichier soit accessible via le système de fichiers (cela permettrait par exemple à un processus tiers de lire et écrire dans la mémoire partagée en lisant et écrivant dans le fichier), il est recommandé de créer le fichier sans entrée de répertoire via l'appel système spécifique à Linux `memfd_create`. La seule autre solution, plus portable mais moins commode, consiste à créer un fichier ordinaire, à le supprimer (tant qu'il est ouvert, il continuera à exister même sans entrée de répertoire) puis à vérifier que personne d'autre n'a eu le temps de l'ouvrir, avant d'effectuer la projection en mémoire. Dans tous les cas, il faut trouver un moyen de partager le descripteur de fichier avec les autres processus devant partager la zone de mémoire. Cela peut se faire par héritage (les processus-fils héritent des descripteurs de fichier de leur parent) ou en transmettant le descripteur via une méthode particulière des sockets.

L'interface System V possède des appels système dédiés pour la mise en place de mémoire partagée et leur détachement. La zone de mémoire n'est pas identifiée par un fichier, mais par une clé d'IPC, une sorte d'identifiant que tous les processus désireux de partager la zone de mémoire doivent connaître. Les opérations à accomplir sont :

1. appel à `shmget` pour obtenir l'identifiant d'une zone de mémoire partagée System V (et la créer si elle n'existe pas encore) ;
2. appel à `shmat` pour attacher cette zone de mémoire partagée dans l'espace d'adresse.

Détacher la zone de mémoire partagée se fait via l'appel système `shmdt` et sa destruction par `shmctl` en passant la commande `IPC_RMID`.

Files de messages

Les files de messages sont des conteneurs d'information conceptuellement semblables à des boîtes aux lettres. Lorsqu'un processus ouvre une file de messages, c'est comme s'il recevait une clé pour la boîte aux lettres. Il peut à tout moment la consulter, lister le courrier déposé à l'intérieur, déposer ou récupérer une lettre. Contrairement aux tubes qui sont strictement séquentiels, les messages sont tous marqués par un entier dont la sémantique est laissée au choix des processus collaborant avec la même file de messages. Il est possible pour un processus de ne lire que les messages ayant un certain identifiant ou bien ayant un identifiant inférieur ou égal à une certaine valeur. Typiquement, ce numéro permet d'implémenter des catégories de messages, ou encore des priorités.

Le fonctionnement des files de messages System V ressemble un peu à celui des mémoires partagées System V. L'appel système `msgget` permet d'obtenir l'identifiant d'une file de messages et au besoin de la créer. Ensuite, les appels système `msgsnd` et

`msgrcv` permettent respectivement d'envoyer et de recevoir des messages via la file. Enfin, l'appel système `msgctl` avec l'opération `IPC_RMID` détruit une file de messages.

Les files de messages System V sont définies par une structure spécifique : `struct msg_queue` [97, `include/linux/msg.h`].

```
struct msg_queue {
  struct kern_ipc_perm q_perm; /* Permission de la file de messages */
  time_t q_stime; /* Horodatage du dernier envoi de message */
  time_t q_rtime; /* Horodatage de la dernière réception de message */
  time_t q_ctime; /* Horodatage du dernier changement de la file */
  unsigned long q_cbytes; /* Occupation de la file en octets */
  unsigned long q_qnum; /* Nombre de messages dans la file */
  unsigned long q_qbytes; /* Taille maximale de la file en octets */
  pid_t q_lspid; /* Identifiant du dernier processus à avoir écrit dans la file */
  pid_t q_lrpid; /* Identifiant du dernier processus à avoir lu dans la file */

  struct list_head q_messages; /* Nombre de messages dans la file */
  struct list_head q_receivers; /* Nombre de récepteurs */
  struct list_head q_senders; /* Nombre d'émetteurs */
};
```

Les files de messages POSIX sont implémentées très différemment et constituent en réalité un type d'i-nœud particulier. Les files ont cependant une interface un peu étrange. La plupart des opérations sur les files, comme la création et la destruction, s'appuient sur le système de fichiers virtuel mais la lecture et l'écriture sont implémentés via deux appels système dédiés : `mq_timedreceive` et `mq_timedsend`. La lecture directe via `read` renvoie quelques informations sur la file, mais pas les messages qu'elle stocke. Les files de messages sont représentées par la structure `struct mqueue_inode_info` [97, `ipc/mqueue.c`].

```
struct mqueue_inode_info {
   [...]
  struct inode vfs_inode; /* I-nœud de la file de message. Ayant l'adresse de la
    structure d'i-nœud il est possible de retrouver l'adresse de la structure de file de messages
    avec de l'arithmétique de pointeurs assez simple. */
  wait_queue_head_t wait_q; /* File d'attente pour les processus souhaitant être
    prévenus de l'arrivée de messages */
  struct rb_root msg_tree; /* Tous les messages de la file, organisés dans un
    arbre rouge–noir */
   [...]
  struct mq_attr attr; /* Informations sur l'état de la file : nombre de messages,
    etc. */
   [...]
  struct user_struct *user; /* Utilisateur ayant créé la file, pour les quotas */
   [...]
  struct ext_wait_queue e_wait_q[2]; /* Files d'attente pour les processus
    attendant que la file se vide ou se remplisse respectivement */
  unsigned long qsize; /* Taille totale en octets de la place occupée par les
    messages */
};
```

Les messages sont représentés par une structure `struct msg_msg` définie dans `include/linux/msg.h`, à l'origine pour les files System V mais réutilisée pour les files POSIX. La structure `struct msg_msg` constitue un conteneur d'information. On ne peut pas prédire à l'avance combien de temps un message va passer dans la file avant d'être reçu, ni même s'il va l'être et encore moins par qui. En effet, les files de messages sont persistantes et un message peut donc être reçu par un processus longtemps après la mort de son émetteur (qui lui-même a pu, de façon similaire, l'émettre avant que le récepteur n'existe).

```
struct msg_msg {
  struct list_head m_list; /* Entrée dans la liste des messages de la file */
  long m_type; /* Type de message, la sémantique en est libre */
  size_t m_ts; /* Taille du contenu du message */
   [...]
  void *security; /* Structure de sécurité LSM */
   /* Le message suit sur le reste de la page de mémoire où est allouée la structure. */
};
```

Sockets réseau

Les *sockets* sont des interfaces génériques représentant un point de communication bidirectionnelle. Une socket peut être connectée à une autre sur la même machine, formant de ce fait un canal de communication un peu similaire à deux tube de sens opposés, ou bien à un port sur une machine distante via le réseau, ou encore à une interface à l'intérieur du noyau. Il existe de nombreux types de sockets en raison de la diversité des protocoles réseau existants mais il est possible de distinguer trois grandes catégories.

— Les sockets de domaine UNIX, originaires de BSD. Ces sockets ne peuvent servir qu'à des communications sur deux processus de la même machine. Elles sont en fait implémentées comme des tampons de mémoire du noyau.

— Les sockets de domaine INET. Ces sockets implémentent l'IP, le protocole de communication entre machines connectées à l'Internet. On peut assimiler à ces sockets toutes celles qui implémentent un protocole inter-machines comme le Bluetooth par exemple. Ces sockets peuvent tout à fait servir à la communication entre deux processus de la même machine, mais avec des performances moindres que d'autres IPC.

— Les sockets spéciales, qui servent à donner un point d'accès dans l'espace utilisateur à des fonctions implémentées dans le noyau, pour avoir accès au matériel par exemple. Les sockets Netlink par exemple permettent de recevoir des notifications d'événements du noyau, en particulier ceux relatifs au réseau, ou d'implémenter de petites routines recueillant des statistiques sur les objets du noyau. Les sockets ALG permettent de bénéficier de l'implémentation de primitives cryptographiques dans le noyau, ce qui est particulièrement intéressant sur les machines disposant d'un coprocesseur cryptographique.

Les sockets sont implémentées dans deux structures principales différentes. La première, `struct socket`, représente l'aspect « fichier » de la socket. En effet, les sockets sont des objets du système de fichiers virtuel qui sont manipulés via des descripteurs de fichiers. Les appels systèmes `read`, `write`, `mmap` sont notamment utilisables. Quelques

appels système comme `send`, `recv` sont fournis car standards, bien qu'ils soient un peu redondants sous Linux. Enfin, quelques appels système comme `bind`, `listen`, `accept` sont nécessaires pour utiliser les sockets et n'ont pas de contrepartie chez les autres types de fichiers. La structure `struct socket` est définie dans `include/linux/net.h`.

```
struct socket {
  socket_state state; /* État de la socket (connectée, en attente de connection,
    etc.) */
   [...]
  short type; /* Type de la socket (mode paquet, mode flux, mode brut, etc.) */
   [...]
  struct socket_wq __rcu *wq; /* File d'attente d'usage variable selon les
    sockets */
  struct file *file; /* Pointeur vers le descripteur de fichier (utile pour certaines
    opérations, en particulier la destruction de la socket et la libération des ressources) */
  struct sock *sk; /* Représentation « côté réseau » de la socket */
  const struct proto_ops *ops; /* Opérations supportées par la socket */
};
```

La deuxième structure de donnée est `struct sock` dont une instance fait partie de chaque structure `socket`. Cette structure est le pendant côté réseau de la structure `struct socket`. Elle est définie dans `include/net/sock.h` et est particulièrement longue. C'est dans cette structure que se trouve le champ `sk_security` qui peut être utilisé par les modules LSM pour attacher aux sockets des attributs de sécurité en plus de ceux de l'i-nœud. La structure `sock` ne dépend pas du protocole réseau. Chaque protocole étend cette structure avec les champs dont il a besoin, en définissant une nouvelle structure dont `sock` est membre.

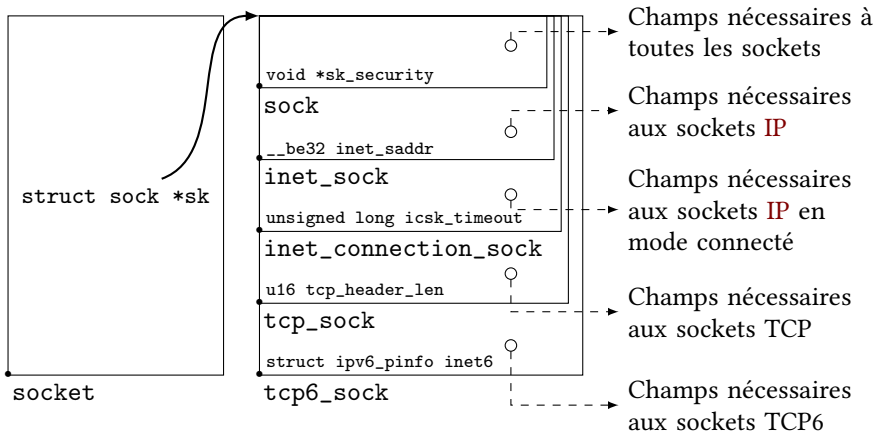


FIGURE 3.1 – Imbrications et liens entre les structures des sockets TCP–IPv6.

On peut prendre comme exemple la structure `tcp6_sock` présentée en figure 3.1. La structure `tcp6_sock` représentant une socket TCP sur IPv6 contient comme premier champ une instance de `struct tcp_sock` représentant une socket TCP générique. Le premier champ de cette structure est une instance de `struct inet_connection_sock` représentant une socket IP fonctionnant en mode connecté (par opposition aux sockets pouvant fonctionner sans connexion explicite, par exemple celles utilisant le protocole

UDP). Cette structure contient à son tour comme premier champ une instance de `struct inet_sock` représentant une socket IP générique. Cette structure contient finalement comme premier champ une instance de `sock`. Cet assemblage a une conséquence très pratique et très utile : si l'on possède un pointeur sur une structure `struct tcp6_sock` , comme toutes les structures imbriquées sont placées au début de leur structure « contenante », ce même pointeur peut donc être converti trivialement en pointeur de chacun des autres types sans changer de valeur. Les structures les plus spécialisées sont donc des extensions des structures génériques. C'est une sorte d'héritage ad hoc, dans un langage qui n'est pas orienté objets.

3.3 Conclusion

Ce chapitre d'exposition ne présente pas une contribution mais plutôt le fruit de nos découvertes dans l'étude du noyau Linux. Tandis que la plupart des ouvrages portant sur le noyau Linux — comme les excellents *Understanding the Linux Kernel* [6] et *Linux Kernel Development* [54] — traitent des structures de données du noyau du point de vue d'un programmeur système débutant essayant de développer son premier noyau Linux personnalisé, nous avons choisi le point de vue atypique des flux d'information. Nous avons présenté comment les abstractions communément appelées « processus », « *thread* », « mémoire » ou encore « fichier » sont en réalité implémentées dans le noyau. Cette présentation sert deux objectifs. Le premier, évident, vise à éclaircir ce que nous entendons par le terme générique de « conteneur d'information ». Le second, indirect, vise à mettre en garde le lecteur contre la tentation de reléguer les problèmes d'implémentation du suivi de flux au second plan. En effet, le code d'un noyau de système d'exploitation, et en l'occurrence de Linux, présente une complexité spécifique et on ne peut pas réellement présupposer de ce qu'il sera possible d'implémenter en termes de mécanisme de sécurité sans avoir une connaissance fine des entrailles du système. Le chapitre prochain poursuit cette idée en présentant notre première contribution : un moyen d'extraire des modèles du code du noyau Linux à des fins d'analyse et de visualisation, qui nous a été grandement utile pour approfondir notre compréhension du code de Linux.

Chapitre 4

Analyse du noyau Linux assistée par GCC

Afin de vérifier que le *framework* LSM est bien à même de servir à l'implémentation correcte de moniteurs de flux d'information, il est important de s'appuyer sur des méthodes formelles. Cependant, les analyses statiques devant porter sur le noyau Linux, cela pose des difficultés spécifiques. Nous devons en particulier être à même de produire des modèles représentatifs et exploitables du code capturant la position des crochets fournis par le *framework* LSM. Dans ce chapitre, nous décrivons la suite d'outils Kayrebt que nous avons développée à cet effet. Nous avons également cherché dans ce travail à produire des outils génériques et réutilisables également pour d'autres objectifs, comme la compréhension du code de Linux (ce qui nous a coûté quelques efforts), ou bien encore d'autres bases de code que celle du noyau Linux.

4.1 Utilisation du compilateur pour produire des modèles du code du noyau Linux

4.1.1 Particularités du code du noyau Linux

Le noyau Linux présente plusieurs caractéristiques qui rendent l'analyse statique difficile. L'aspect le plus évident est la taille du code source. Le noyau Linux compte, dans sa dernière version, environ quinze millions de lignes de code réparties sur quarante-trois mille fichiers. Bien entendu, aucun noyau compilé et installé n'utilise l'intégralité de ce code. La majorité de celui-ci constitue en réalité les pilotes de périphériques et les définitions des systèmes de fichiers. Cela contribue au vaste support matériel du noyau, mais chaque machine n'a besoin en réalité que d'une partie de tout ce code. Les fonctionnalités essentielles du noyau (l'ordonnancement, le système de fichiers virtuel, la pagination, la pile réseau, etc.) sont fournies par une petite fraction du code total. Néanmoins, même cette simple fraction constitue à elle seule un code de taille impressionante. La deuxième particularité du code est son langage source. En effet, le noyau Linux est codé majoritairement en C, mais pas en C du standard ANSI [92]. Les extensions du compilateur GCC sont explicitement permises dans le noyau [77]. Jusqu'aux dernières versions, il était d'ailleurs impossible de compiler Linux avec un autre compilateur que GCC. La situation a quelque peu changé depuis que la suite de

compilateur LLVM a pris de l'importance, mais le noyau ne permet toujours pas la compilation avec un autre compilateur que GCC sans quelques modifications. En plus du C, une partie du code du noyau est écrit en assembleur, dans la syntaxe propre à GCC. Ce code ne peut naturellement pas s'analyser de la même manière que le C, ce qui complique l'écriture d'analyses statiques.

Ensuite, le noyau présente la difficulté d'être massivement parallélisé. En effet, depuis la disparition du *Big Kernel Lock* en 2011 [3], il n'existe plus de zone critique dans le noyau, à l'intérieur de laquelle un *thread* pourrait être absolument seul à s'exécuter, hormis au démarrage. À tout moment, des dizaines de *threads* différents sont en activité. Comme le noyau s'exécute la plupart du temps en mode préemptif[1] et depuis la généralisation des architectures multi-processeurs, il doit faire face à la fois au faux et vrai parallélismes à tout point du code. C'est une difficulté pour les outils d'analyse statique car de nombreuses conditions de concurrence peuvent survenir et les contextes dans lesquelles elles peuvent se déclencher ne sont pas toujours facilement capturables.

Le noyau présente également la caractéristique d'avoir plusieurs points d'entrée. En effet, tandis qu'un programme classique démarre par convention son exécution depuis une fonction appelée `main` dans un programme C, elle-même appelée depuis `__start`, n'importe quel appel système ou gestionnaire d'interruption constitue pour le noyau le début d'un chemin d'exécution. Cela est différent d'un serveur qui attend une connexion pour démarrer un nouveau processus afin de servir la requête du client. En effet, dans le cas du noyau, un *thread* de l'espace utilisateur qui réalise un appel système *devient*, jusqu'au retour de l'appel système, un *thread* de l'espace noyau. Il incombe en particulier au code du noyau de supprimer avant le retour en espace utilisateur tout ce qui, dans le contexte d'exécution du *thread*, pourrait révéler de l'information sur le noyau ; hormis, naturellement, la sortie attendue de l'appel et les effets de bords prévus dans sa spécification. Le noyau étant à l'interface entre le matériel et les processus de l'utilisateur, il contient également le code permettant de traiter les interruptions matérielles et logicielles du système. Ce code constitue également un point d'entrée du noyau. Par exemple, lorsqu'un processus effectue une division par zéro, une interruption est immédiatement levée par le processeur. La tâche en cours d'exécution est alors suspendue et le pointeur d'instruction saute à l'adresse d'une fonction du noyau, programmée au démarrage. Le code du noyau en charge du traitement de cette exception est donc exécuté « par surprise » et doit réagir pour traiter l'interruption (en l'occurrence, en délivrant un signal SIGFPE au processus fautif) et repasser la main à l'espace utilisateur. Pour appliquer des analyses classiques, il faut dans la pratique considérer chaque appel système et gestionnaire d'interruption comme un exécutable propre, à ceci près que le noyau maintient un très lourd état partagé par tout le code du noyau. Notons cependant que cette particularité d'avoir plusieurs points d'entrée n'est pas unique au noyau. Les applications Android ont également un point d'entrée par fenêtre graphique et par service, l'équivalent de plusieurs fonctions `main` en quelque sorte.

Le dernier facteur de complexité est le caractère dynamique du noyau. Le modèle de développement du noyau est maintenant rodé depuis plusieurs années. Dès qu'une version du noyau est publiée commence aussitôt la préparation de la suivante. Les deux premières semaines suivant la publication d'une version sont ainsi appelées période d'incorporation (*merge window* en anglais), car c'est durant ce laps de temps que de

1. Mode dans lequel un *thread* est susceptible de voir son exécution être interrompue à tout moment afin de laisser un autre *thread* s'exécuter, par opposition au mode coopératif où les *threads* relâchent volontairement le CPU.

nouvelles fonctionnalités peuvent être soumises à l'examen attentif de Linus Torvalds, mainteneur-en-chef de Linux, pour leur ajout au noyau. Les implémentations de ces fonctionnalités ont en général séjourné pendant quelque temps dans des branches de tests des dépôts de développement. Ensuite, une fois la période d'incorporation passée, la version en préparation est stabilisée et la première version candidate à la publication (*release candidate* en anglais) est produite. L'activité suivante des développeurs est de tester intensivement et extensivement cette version pendant les semaines à venir. Toutes les semaines, une nouvelle version candidate est produite, incluant les modifications apportées durant la semaine. Seules les corrections de problèmes sont acceptées durant cette phase et il est rigoureusement interdit de proposer de nouvelles fonctionnalités. Il faut en général de six à huit versions candidates pour que le volume de correctifs diminue suffisamment et que Torvalds ne juge le noyau suffisamment stabilisé pour publier une nouvelle version, à la suite de quoi le cycle recommence avec une nouvelle période d'incorporation. Il s'ensuit donc qu'une nouvelle version du noyau est publiée environ tous les deux mois à deux mois et demi, ce qui constitue un rythme assez soutenu. Les analyses faites sur le noyau se doivent d'être reproductibles et réemployables d'une version à la suivante pour être utiles.

En revanche, par rapport au code d'un programme commun, le noyau présente certains avantages facilitant l'écriture d'analyses statiques. En premier lieu, seules des variables entières sont utilisées. En effet, l'arithmétique sur les nombres réels (utilisant le codage IEEE 754) n'est pas possible dans le noyau en règle générale. Cela est dû au fait que ce type d'arithmétique requiert un jeu d'instructions — et en général une unité de calcul — dédié, qui n'est pas présent sur toutes les architectures supportées par Linux, en particulier les processeurs pour systèmes embarqués. Même sur les architectures le possédant, il faut arbitrer son utilisation entre les *threads* du noyau et ceux de l'espace utilisateur, ce qui représente un coût certain. Les types flottants forment donc une catégorie de variables qu'il n'est pas nécessaire de considérer.

Enfin, le noyau ne dépend d'aucune base de code tierce. Il inclut y compris sa propre bibliothèque standard. En effet, il est impossible pour un système d'exploitation de pouvoir dépendre à l'exécution du chargement d'une bibliothèque. On a donc la garantie lors de l'analyse statique d'avoir une vue complète sur l'intégralité du code à analyser, sans mauvais surprise possible [2].

4.1.2 Utilité du compilateur pour construire des modèles

Le compilateur a pour tâche de transformer le code source du noyau, écrit dans un dialecte du langage C et en assembleur (dont la forme dépend de l'architecture cible), en un exécutable, également appelé *image*, le noyau. L'image noyau est un produit de compilation atypique. Il est, par certains aspects, classique : il s'agit somme toute d'une grosse archive statique et sa chaîne de compilation n'a guère changé depuis des années. Par d'autres aspects, il est très élaboré. En effet, le noyau supporte par exemple le chargement dynamique de code, sous la forme de modules. De plus, le processus de compilation du code est extraordinairement paramétrable. En effet, Linux supporte de nombreuses architectures cibles et de nombreuses fonctionnalités du noyau peuvent être sélectionnées ou non pour faire partie du noyau. On peut ainsi sélectionner les systèmes de fichiers et les pilotes de périphériques que l'on souhaite pouvoir utiliser,

2. Il est cependant possible de charger du code externe au noyau officiel, sous la forme de modules. C'est utile pour tester de nouvelles fonctionnalités avant de les proposer à l'inclusion dans le noyau officiel ou bien pour fournir le support d'un périphérique sans donner le code source du pilote. Le noyau est dans ce cas dit « sali » (*tainted* en anglais). Dans nos analyses, nous n'avons travaillé que sur des noyaux propres.

ou encore ne pas inclure le support du protocole IPv6 dans l'image finale par exemple. Lire le code est donc un exercice délicat car il faut être capable de se représenter mentalement quelles sont les lignes de code qui sont pertinentes. En revanche, le compilateur est toujours capable de faire la distinction entre le code sélectionné pour la compilation et le code laissé de côté.

Étant donné la taille massive de la base de code et ses difficultés intrinsèques, il apparaît nécessaire de produire de manière automatique et reproductible les modèles qui serviront de support à l'analyse du noyau Linux. La tâche du compilateur est précisément de donner une sémantique au code. Développer un nouvel analyseur syntaxique et sémantique du langage C pour extraire des visualisations est donc inutile, voire néfaste, car étant donné que le C n'a pas de sémantique formelle, rien ne garantit qu'une autre analyse que celle de GCC (qui est le compilateur « officiel » du noyau, en tout cas, le seul explicitement supporté par la chaîne de compilation de Linux) produira un résultat équivalent. Notre idée de départ est donc que si GCC est capable de produire un exécutable optimisé, alors ses représentations intermédiaires devraient également comporter suffisamment d'information pour implémenter nos propres analyses statiques du code. Cela signifie donc que GCC doit avoir, d'une manière ou d'une autre, des représentations internes pouvant servir à la fois de modèle du code source et de modèle de l'exécutable. Enfin, notre dernière intuition a été la suivante : si les représentations internes du code par GCC peuvent servir de fichier d'entrée à des outils d'analyse statique, alors ces mêmes représentations devraient être exploitables par des humains pour appréhender une base de code telle que celle du noyau. En d'autres termes, nous sommes convaincus du potentiel que recèlent les artefacts de compilation du code en termes de modèles et de visualisations du code.

4.2 Extraction et visualisation de graphes de flot de contrôle avec Kayrebt

Nous avons validé notre hypothèse que le compilateur est l'outil le plus indiqué pour produire des modèles et visualisations du code en implémentant le projet Kayrebt, qui comporte plusieurs composants :

- *Kayrebt::Extractor* est un greffon GCC dont le rôle est d'extraire, pendant la compilation du code source d'une base de code C des graphes de flot de contrôle de chaque fonction.
- *Kayrebt::Callgraphs* est un autre greffon permettant d'extraire des graphes d'appel de fonctions d'une base de code C.
- *Kayrebt::Dumper* est un ensemble de scripts lançant *Kayrebt::Extractor* sur la base de code du noyau Linux, selon différentes options d'extraction.
- *Kayrebt::Globsym* est un ensemble de scripts extrayant depuis la base de code du noyau l'ensemble des fonctions définies dans le noyau associées à leur fichier et ligne de définition. Cette connaissance est extraite des informations de débogage du noyau produites par GCC. La base de données de fonctions produites est exploitée par *Kayrebt::Extractor* afin de produire des graphes enrichies de métadonnées telles que la ligne et le fichier du code source correspondant à chaque nœud.
- *Kayrebt::Viewer* est une interface graphique multi-plateformes permettant la visualisation et la navigation parmi les graphes produits par *Kayrebt::Extractor* pour une base de code donnée, en particulier celle du noyau.

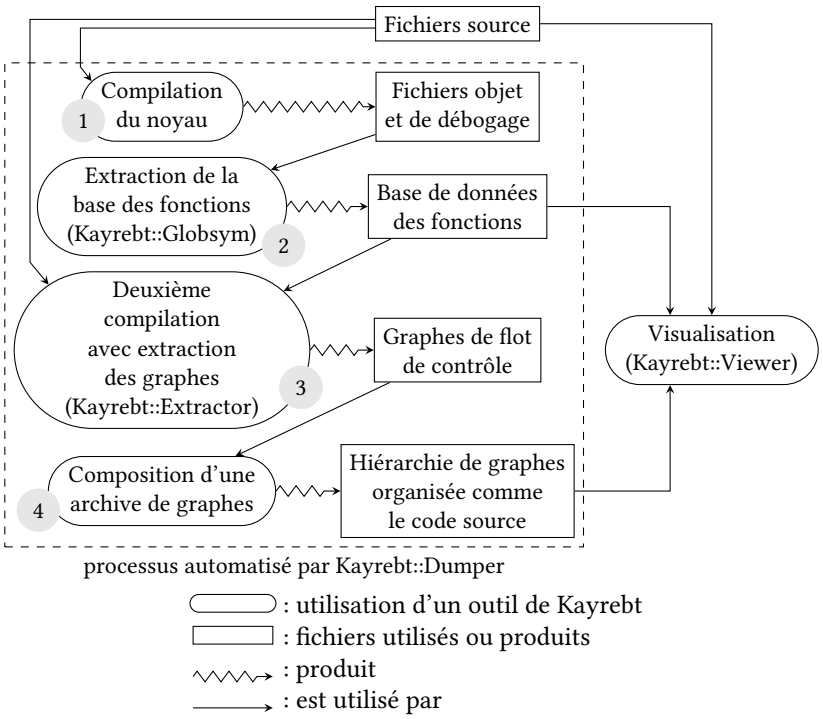


FIGURE 4.1 – Processus d'extraction des graphes de flot de contrôle d'une base de code avec la suite d'outils Kayrebt

La figure 4.1 illustre l'intégration de ces différents outils. On notera que Kayrebt::Callgraphs est un outil à part des précédents.

Nous décrivons dans ce chapitre les outils Extractor, Viewer et Callgraphs.

4.2.1 Kayrebt::Extractor : un greffon d'extraction de graphes de flot de contrôle pour GCC

La suite de compilateurs *Gnu Compilers Collection* supporte l'injection de greffons depuis la version 4.5.0 [88, 89]. Les greffons se présentent sous la forme de bibliothèques pouvant être chargées dynamiquement pendant l'exécution. Les greffons sont passés en paramètre de GCC et il est possible de leur passer toute sorte d'argument. L'emploi des greffons est cependant limité : comme le compilateur est appelé indépendamment sur chaque *unité de traduction*[3], on ne peut pas travailler facilement sur l'ensemble d'une base de code et il faut que les transformations ou manipulations du code soient locales à cette unité de traduction.

Syntaxe des graphes

Extractor produit un graphe par fonction du code. Chaque nœud correspond à une instruction et un arc matérialise le fait qu'une instruction peut en suivre une autre dans le code. Chaque graphe contient un unique nœud nommé « INIT » sans

3. Dans le cas du langage C, une unité de traduction correspond en généralement à un fichier source `.c`.

prédécesseur, correspondant à l'entrée de la fonction ainsi qu'un unique nœud « END » sans successeur, correspondant au retour normal de la fonction. Bien sûr, dans le code source, une fonction peut normalement avoir plusieurs instructions *return*, mais le compilateur GCC produit une instruction fictive unique pour joindre tous ces nœuds à la compilation. Les autres nœuds sans successeurs représentent les crashs délibérés du programme (dans le cas du noyau, on les appelle des *kernel panics*). Il est clair d'après ceci que les graphes ne représentent pas le code dans le langage source. En réalité, on extrait une représentation intermédiaire du code construite par GCC, dans un langage appelé GIMPLE [58]. L'extrait de code 4.1 montre le code de la fonction `vfs_llseek` extraite du noyau Linux. Cette fonction implémente l'appel système `llseek`, qui permet de déplacer la position de lecture dans un fichier. On voit dans l'extrait que dans un premier temps, un pointeur de fonction est déclaré. Ensuite, selon que le fichier `file` considéré (passé en paramètre) supporte l'opération de déplacement de la position de lecture ou non, soit une fonction dépendante du système de fichiers sur lequel `file` réside, soit une fonction générique ne faisant rien est choisie. La fonction sélectionnée est finalement appelée. L'extrait 4.2 représente la même fonction telle qu'interprétée par GCC durant la compilation. GCC est organisée sous forme d'une succession de phases : chaque phase modifie le code pour appliquer une transformation. Cette transformation peut constituer en une optimisation ou bien peut calculer une propriété du code, comme l'allocation de registres, ou encore passer le code d'un langage de représentation à un autre. Enfin, la figure 4.2 présente le graphe construit par Kayrebt::Extractor pour la fonction `vfs_llseek`.

```
1  loff_t vfs_llseek(struct file *file, loff_t offset, int
       whence)
   {
     loff_t (*fn)(struct file *, loff_t, int);

5    fn = no_llseek;
     if (file->f_mode & FMODE_LSEEK) {
       if (file->f_op->llseek)
         fn = file->f_op->llseek;
     }
10   return fn(file, offset, whence);
   }
```

Extrait 4.1 – Code de la fonction `vfs_llseek`

Le code GIMPLE représente par lui-même un graphe de flot de contrôle, à ceci près que ce ne sont pas les instructions unitaires qui forment les nœuds mais les *basic blocks*. Un *basic block* est une séquence d'instructions qui se suivent nécessairement et qui sont terminées par un saut conditionnel vers plusieurs autres *basic blocks*. Comme on peut le voir dans l'extrait 4.2, la fonction contient quatre *basic blocks* délimités par `<bb N>`. On remarque que le *basic block* 4 est vide et suivi inconditionnellement du *basic block* 5. L'intérêt de ce *basic block* est le nœud « PHI » [4], au début du *basic block* 5. Ce type de nœud sert à assigner une variable de manière différente selon le chemin suivi dans la fonction. Il est utilisé lorsque le graphe de flot de contrôle est sous une certaine forme appelée *Static Single Assignment* [18] (détaillée plus loin dans cette section). En l'occurrence ici, `fn_1` reçoit la valeur `no_llseek` si l'exécution vient des *basic blocks* 2 ou 4 et `fn_7` si l'exécution vient du *basic block* 5. Dans nos graphes tels que ceux de la figure 4.2, les nœuds en forme de losange représentent les extrémités

4. On note ces nœuds « ϕ » dans la suite de cette thèse.

de *basic blocks*. Les arcs comportant une garde représentent les sauts conditionnels, la garde donnant la condition de franchissement. Le dernier nœud avant « END », le rectangle dans le graphe, est la valeur de retour de la fonction.

```
vfs_llseek (struct file * file, loff_t offset, int whence)
{
  loff_t (*<T26db>) (struct file *, loff_t, int) fn;
  unsigned int _4;
  unsigned int _5;
  const struct file_operations * _6;
  loff_t _11;

  <bb 2>:
  _4 = file_3(D)->f_mode;
  _5 = _4 & 4;
  if (_5 != 0)
    goto <bb 3>;
  else
    goto <bb 5>;

  <bb 3>:
  _6 = file_3(D)->f_op;
  fn_7 = _6->llseek;
  if (fn_7 != 0B)
    goto <bb 5>;
  else
    goto <bb 4>;

  <bb 4>:

  <bb 5>:
  # fn_1 = PHI <no_llseek(2), no_llseek(4), fn_7(3)>
  _11 = fn_1 (file_3(D), offset_8(D), whence_9(D));
  return _11;

}
```

Extrait 4.2 – Représentation intermédiaire de la fonction `vfs_llseek` produit par GIMPLE

Avantages et inconvénients d'extraire les graphes depuis la représentation interne du compilateur

Le contenu des graphes est suffisamment proche du code d'origine pour qu'on puisse le reconnaître. Cela est dû au fait que le langage C est de relativement bas niveau. Nous avons fait quelques expérimentations en appliquant Kayrebt::Extractor sur du code en C++ et en Java. Ceci est possible car GCC traite tous les langages qu'il accepte en entrée de la même manière : ils sont tout d'abord transformés en sa représentation intermédiaire en langage GIMPLE, qui est celle qu'Extractor manipule. Les résultats se sont montrés beaucoup moins exploitables car le C++ et le Java sont des langages de beaucoup plus haut niveau que le C. Dans notre test — une simple fonction acceptant un entier sur l'entrée standard pour l'afficher en hexadécimal sur la sortie standard — nous avons été surpris par l'apparition subite d'une trentaine

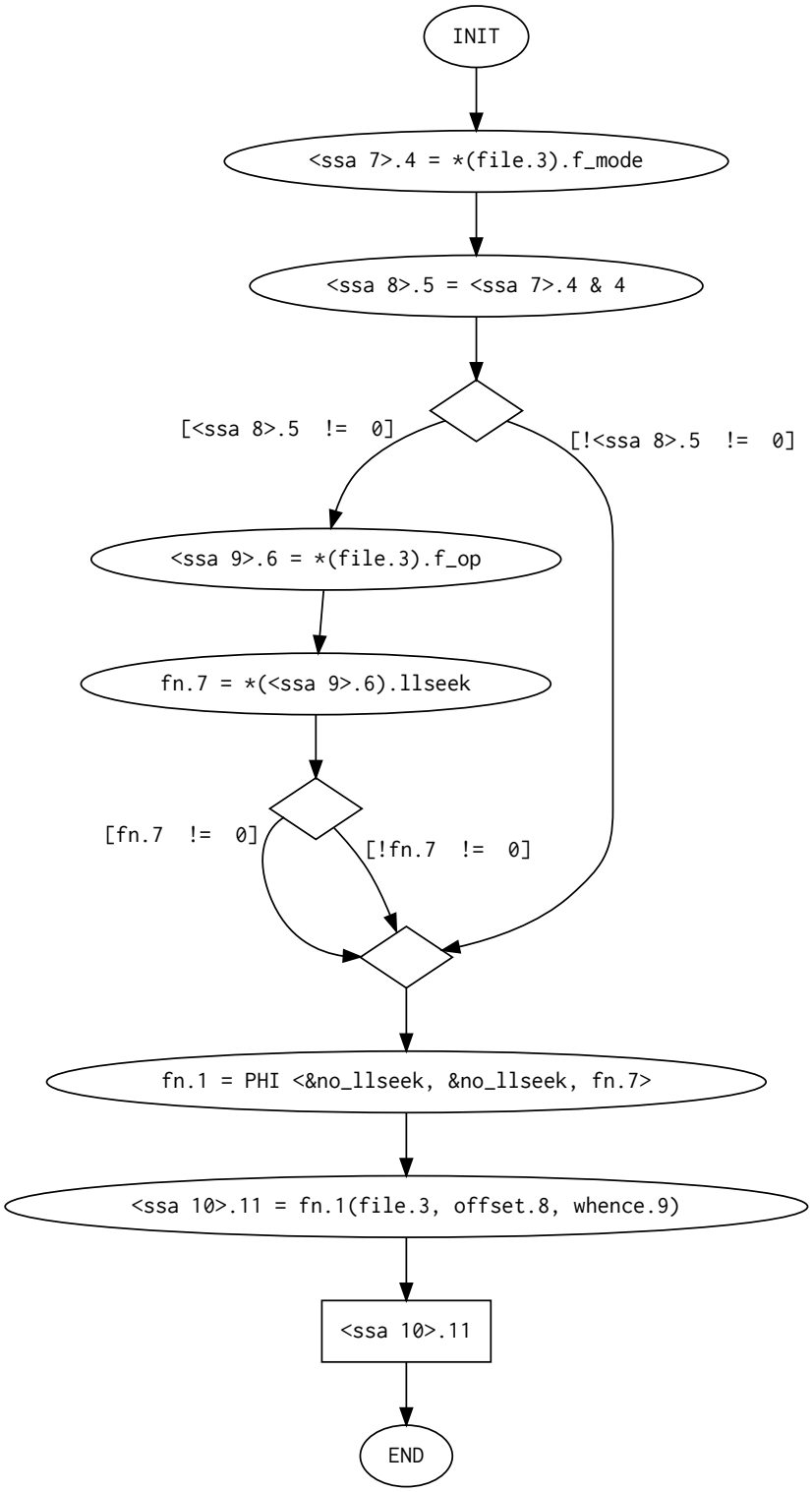


FIGURE 4.2 – Graphe de la fonction `vfs_llseek`

de fonctions aux noms barbares générées par le compilateur, et autant de graphes. Le résultat est difficilement exploitable pour visualiser le code. Il peut cependant se révéler intéressant pour quelqu'un voulant comprendre le mécanisme de compilation d'un langage objet de haut niveau.

Dans le cas du C, on obtient exactement un graphe par fonction écrite, ou un peu moins si le compilateur juge pertinent d'évincer quelques unes des fonctions, et le code est reconnaissable dans le graphe. Il y a cependant des différences frappantes entre le code d'origine et le graphe. En premier lieu, il faut remarquer que, Kayrebt::Extractor étant un greffon du compilateur C, le code qui est compilé n'est pas celui d'origine, mais le code déjà transformé par le *préprocesseur*. Toutes les macros du code d'origine, ici par exemple dans l'extrait 4.1 la macro `FMODE_READ`, ont été remplacées par leur valeur, en l'occurrence 4. Ceci est à la fois un avantage et un inconvénient. En effet, en particulier dans le cas du noyau Linux, les développeurs usent et abusent des macros pour coder des itérateurs sur certaines structures de données, pour compiler du code optionnellement, pour fournir différentes implémentations d'une même fonction selon l'architecture pour laquelle le noyau est compilé, etc. Ces macros peuvent être difficiles à suivre lorsqu'on lit le code source. En revanche, les graphes contiennent exactement le code qui participe à la compilation et donc qui fera partie de l'exécutable final. Les graphes peuvent donc servir à montrer quel code est réellement utilisé. Néanmoins, dans le cas où les macros sont utilisées comme noms symboliques pour cacher des constantes numériques arbitraires, le fait qu'elles disparaissent dans les graphes est plutôt un inconvénient car on perd en lisibilité.

Dans le cas du langage C comme dans l'extrait 4.1, on note également des différences entre le code d'origine et le graphe. Premièrement, dans le graphe et le code sous forme GIMPLE, il y a beaucoup plus de variables que dans le code de base. En effet, le compilateur, parmi ses multiples traitements, simplifie le code de sorte à découper les opérations complexes impliquant plusieurs opérateurs et déréférencements de pointeurs en opérations unitaires. Cela implique de synthétiser des variables supplémentaires. Cet effet est encore renforcé par le fait qu'au point où le graphe est extrait dans la chaîne de compilation, le code est sous la forme *Static Single Assignment* [18], c'est-à-dire que GCC s'applique à versionner les variables selon le point où elles reçoivent une valeur. Ainsi, en principe, chaque variable versionnée connaît un seul point d'affectation (comme s'il s'agissait de constantes), ce qui simplifie certaines optimisations dans le code. Il y a des exceptions à ce principe en particulier dans le cas des variables globales ou volatiles, ce qui explique que GCC applique des optimisations beaucoup moins poussées dans les codes en utilisant. GCC crée des nœuds ϕ pour représenter le fait qu'une certaine variable pourrait avoir des valeurs différentes selon le chemin pris à l'exécution dans la fonction. Dans le code abouti, après la compilation, ces instructions ϕ n'existent pas. Si on désire qu'une certaine variable a ait soit la valeur de b soit la valeur de c selon le chemin pris, il suffit que dans leurs chemins respectifs, b et c représente la même case mémoire. Ainsi, à l'endroit où se trouverait le nœud ϕ, on peut donner à cette case mémoire commune le nom a et on obtient le résultat désiré. La conclusion de ceci est que se servir du compilateur pour extraire des représentations visuelles du code peut conduire à des surprises. En effet, il faut accepter toutes les idiosynchrasies de la représentation intermédiaire produite par le compilateur. Il convient donc de choisir soigneusement à quel point de la chaîne de compilation on désire insérer Kayrebt::Extractor. Dans notre cas, nous avons choisi le point de la chaîne de compilation où le code est le plus optimisé tout en gardant à disposition le graphe de flot de contrôle maintenu par GCC. En effet, dans les dernières phases de compilation, GCC abandonne cette représentation.

Utilisation de Kayrebt::Extractor

GCC est nativement capable de produire des graphes de flots de contrôle sous forme graphique, au format Graphviz [27]. Kayrebt::Extractor produit également ses graphes dans ce format. L'extrait 4.3 donne la définition du graphe de la figure 4.2. Néanmoins, les graphes produits présentent des différences. En premier lieu, nos graphes sont enrichis grâce à la base des fonctions produites par Kayrebt::Globsym. En effet, chaque nœud correspondant à un appel statique de fonction contient en argument un lien hypertexte vers le graphe de la fonction appelée [5]. La base de fonctions nous permet de distinguer les fonctions dites *static* des fonctions *globales*. En C, il ne peut y avoir qu'une seule fonction d'un nom donné dans une unité de traduction. Les fonctions globales sont communes à toutes les unités tandis que les fonctions statiques sont locales à leur unité de traduction. Dans une même base de code, il peut donc y avoir plusieurs fonctions statiques du même nom dans différents fichiers. La base des fonctions nous permet dans tous les cas de faire correspondre un nœud d'appel de fonction d'un graphe au graphe de la fonction correspondante.

```
digraph G {
graph [file="fs/read_write.c",line=252,parameters="offset,
    whence"]
29[shape="ellipse",label="INIT",type=init];
31[shape="ellipse",label="<ssa 7>.4 = *(file.3).f_mode",line
    =256,filename="fs/read_write.c",type=assign];
32[shape="ellipse",label="<ssa 8>.5 = <ssa 7>.4 & 4",line
    =256,filename="fs/read_write.c",type=assign];
33[shape="diamond",label="",line=256,filename="fs/read_write
    .c",type=cond];
35[shape="ellipse",label="<ssa 9>.6 = *(file.3).f_op",line
    =257,filename="fs/read_write.c",type=assign];
36[shape="ellipse",label="fn.7 = *(<ssa 9>.6).llseek",line
    =257,filename="fs/read_write.c",type=assign];
37[shape="diamond",label="",line=257,filename="fs/read_write
    .c",type=cond];
39[shape="diamond",label=""];
40[shape="ellipse",label="fn.1 = PHI <&no_llseek, &no_llseek
    , fn.7>",type=phi];
41[shape="ellipse",label="<ssa 10>.11 = fn.1(file.3, offset
    .8, whence.9)",line=260,filename="fs/read_write.c",type=
    call];
42[shape="rect",label="<ssa 10>.11",line=260,filename="fs/
    read_write.c",type=return];
43[shape="ellipse",label="END",type=end_of_activity];
31->32 [label=""];
32->33 [label=""];
35->36 [label=""];
36->37 [label=""];
39->40 [label=""];
40->41 [label=""];
41->42 [label=""];
33->39 [label="[!<ssa 8>.5  !=  0]"];
```

5. Dans le graphe donné en exemple, le seul appel de fonction est dynamique, c'est-à-dire qu'il est réalisé au travers d'un pointeur de fonction ; dans ce cas, il est impossible de connaître statiquement la fonction appelée donc Extractor ne place pas de lien hypertexte.

```
37->39 [label="[fn.7  !=  0]"];
42->43 [label=""];
29->31 [label=""];
33->35 [label="[<ssa 8>.5  !=  0]"];
37->39 [label="[!fn.7  !=  0]"];
}
```

Extrait 4.3 – Définition du graphe de la fonction `vfs_llseek` au format Graphviz

Chaque nœud contient également en attribut le type d'instruction auquel il correspond, ainsi que le fichier source et la ligne de l'instruction du code source à laquelle il correspond. On extrait cette connaissance des informations de débogage et de diagnostic du compilateur.

Enfin, il est possible d'attribuer des catégories arbitraires aux nœuds et aux arcs des graphes et de leur associer des commandes de formatage Graphviz. Nous avons utilisé cette fonctionnalité pour produire des visualisations où les nœuds correspondant à des crochets LSM sont mis en évidence dans les graphes avec une couleur de fond particulière. Ce mécanisme permet d'appréhender d'un coup d'œil l'utilisation des fonctions d'une certaine API dans une base de code.

Il est nécessaire de configurer Kayrebt::Extractor pour utiliser ces fonctionnalités. Cela se fait à l'aide d'un fichier au format YAML dont l'extrait 4.4 donne un exemple.

```
general:
  greedy: 0
  url:
    dbfile: 'my_db.sqlite'
    dbname: 'symbols'
  categories:
    1: 'bgcolor=blue'
    2: 'textcolor=red'

source_file1.c:
  functions: ['function1','function2']
  match:
    '(k|m)alloc.*': 1

source_file2.c:
  functions: ['one_more_function']
  start_match:
    '.*spin_lock.*': 2
  end_match: ['spin_unlock']
```

Extrait 4.4 – Exemple de configuration

On voit dans cet extrait toutes les possibilités de configuration. Le fichier est séparé en sections, une par fichier à compiler, plus une section spéciale *general* qui s'applique globalement. Si l'on souhaite extraire tous les graphes possibles d'une base de code, la configuration minimale consiste à avoir seulement la section *general* avec la valeur *greedy* à 1. Autrement, il faut spécifier, pour chaque fichier, la liste des fonctions dont on veut extraire le graphe. La section « url » donne le nom du fichier et de la base de données produite par Kayrebt::Globsym. Les définitions des catégories donnent les attributs supplémentaires des nœuds concernés. Les sections « source_file1.c » et « source_file2.c » montrent deux exemples de classification des nœuds dans des catégories. Dans « source_file1.c », on assigne à tous les nœuds correspondant à l'expression régulière `(k|m)alloc.*` (c'est-à-dire les fonctions d'allocation de la mémoire

du noyau Linux) la catégorie 1. Dans « source_file2.c », on associe à tous les nœuds situés entre un nœud `spin_lock` et un nœud `spin_unlock` (qui correspondent à des fonctions de synchronisation de *threads* dans le noyau) la catégorie 2. Ce type de visualisation permet dans cet exemple de voir le code qui est exécuté entre deux instructions de verrouillage d'une certaine structure de donnée, ce qui peut correspondre à une section critique particulièrement intéressante du code. Sur l'ensemble des graphes d'une base de code, on peut ainsi apprécier d'un seul coup d'œil si le code est bien parallélisé ou bien s'il est contraint par trop de verrous.

4.2.2 Kayrebt::Viewer : une interface de visualisation des graphes produits par Extractor

La collection de graphes produite par Kayrebt::Extractor est déjà en soi intéressante, mais visualiser les graphes individuellement présente un intérêt limité dans une base de code aussi grande que le noyau. Il est plus intéressant de pouvoir naviguer d'un graphe à l'autre, de suivre les appels de fonction, etc. Pour cette raison, nous avons conçu les autres outils de la suite Kayrebt, et en particulier Viewer. Kayrebt::Viewer est essentiellement une interface graphique pour parcourir les graphes de flots de contrôle. Il dépend également des fichiers source, ainsi que de la base de fonctions construite par Kayrebt::Globsym. À l'ouverture du logiciel, on doit donner le chemin du dossier contenant la base de code, le fichier contenant la base des fonctions et enfin le chemin du dossier des graphes. Ce dernier doit être organisé comme le dossier des sources, à ceci près que les fichiers « .c » du code source deviennent des dossiers contenant les graphes des fonctions du fichier source au format Graphviz. Il y a un fichier graphe par fonction, portant le nom de celle-ci.

La figure 4.3 donne un aperçu de l'interface. Le panneau central montre le graphe en cours de visualisation. On peut manipuler cette vue de plusieurs manières. On peut zoomer et dézoomer avec la molette de la souris. Les graphes peuvent être de taille très variable, entre moins d'une dizaine et plus d'une centaine de nœuds. Survoler un nœud avec la souris l'affiche en rouge, ainsi que tous les nœuds atteignables depuis le nœud survolé. Double-cliquer sur un nœud le supprime. On supprime également du même coup tous les nœuds et arcs du graphe pour lesquels il n'existe plus de chemin depuis le nœud « INIT » du graphe. Ces fonctionnalités permettent de filtrer aisément certaines branches des graphes qui ne sont pas pertinentes pour la visualisation.

À chaque fois qu'un graphe est ouvert, le panneau de droite affiche le code source de la fonction correspondante. On peut ainsi examiner chaque fonction et son graphe en vis-à-vis. Lorsqu'une fonction comporte beaucoup de macros ou de parties compilées optionnellement, cela permet de suivre le flot de contrôle de manière fiable et aide à la compréhension du code.

Il existe plusieurs moyens de passer d'un graphe à un autre dans le panneau central. On peut tout d'abord se servir du panneau de gauche. La partie haute présente trois onglets. Le premier est une table représentant de manière directe le contenu de la base des fonctions. On peut filtrer cette table par des expressions régulières sur les noms de fonctions ou sur les fichiers et dossiers du code source. Le deuxième onglet donne l'historique des graphes ouverts dans la session d'utilisation du Viewer courante. Enfin, le dernier onglet (ouvert dans la figure 4.3) est l'arborescence du dossier des graphes. En connaissant l'organisation des sources, on peut rapidement trouver le graphe d'une fonction particulière. Quel que soit l'onglet ouvert dans la partie haute, double-cliquer sur le nom d'une fonction ouvre son graphe.

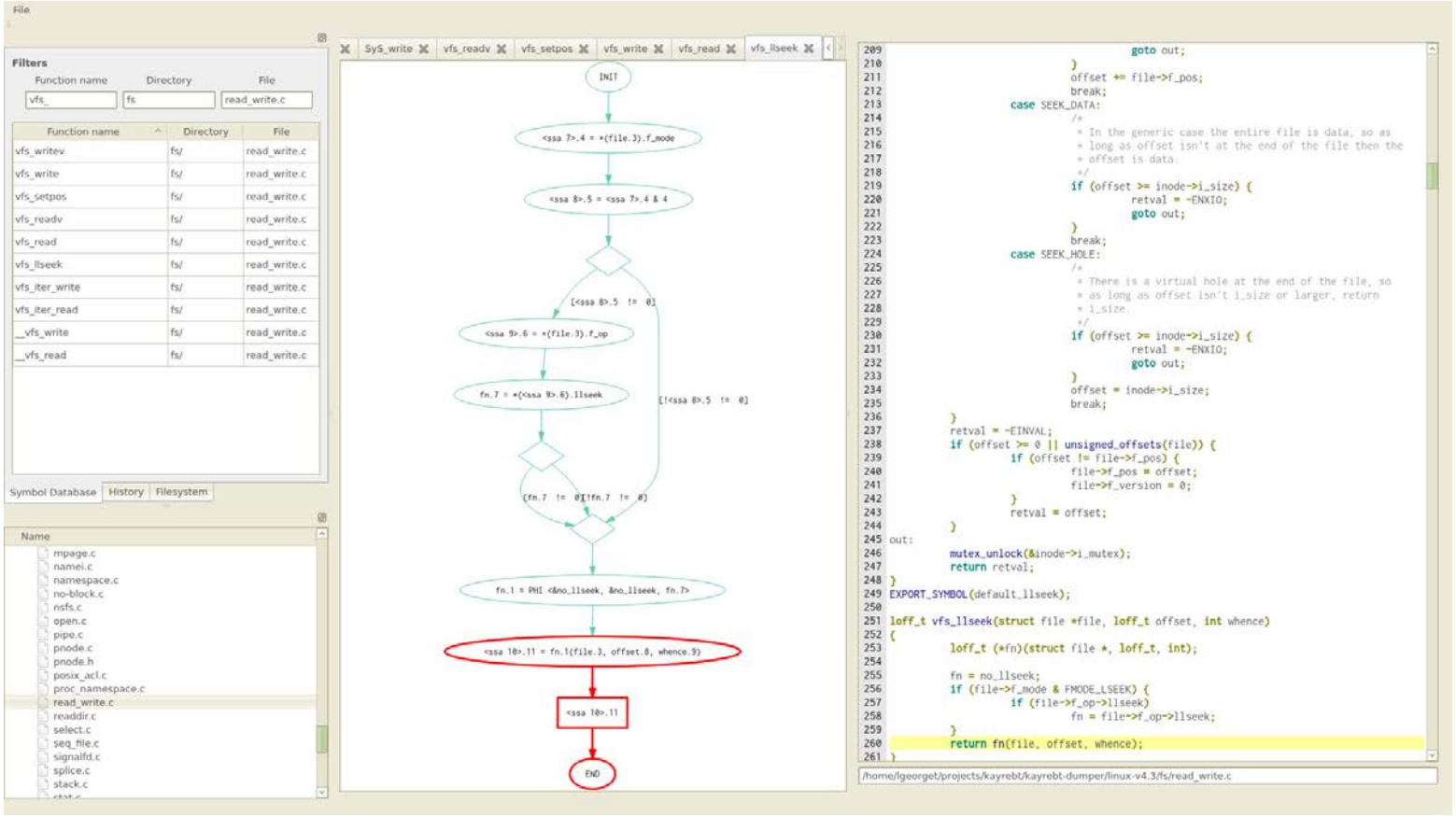


FIGURE 4.3 – Interface de Kayrebt::Viewer

La partie basse du panneau de gauche est l'arborescence du code source. Cette vue a deux utilités. Premièrement, à chaque fois qu'un graphe est sélectionné dans un onglet de la partie haute, le fichier contenant la définition de la fonction correspondant est mis en évidence dans l'arborescence de la partie basse. Dans une base de code particulièrement grande comme le noyau Linux, cela est particulièrement utile pour prendre en main l'organisation des sources. Ensuite, sélectionner un fichier dans cette arborescence met à jour les filtres du premier onglet de la partie haute avec le nom du fichier et de son dossier. On peut ainsi parcourir rapidement la liste des fonctions de chaque fichier.

4.2.3 Kayrebt::Callgraphs : un greffon pour produire des graphes d'appel

Kayrebt::Callgraphs produit des graphes d'un autre type que ceux produits par Extractor. Tandis que les graphes de flot de contrôle permettent d'apprécier l'organisation et les chemins d'exécutions à l'intérieur d'une fonction, les graphes d'appel de fonction permettent de comprendre comment les fonctions s'appellent les unes les autres. Tout comme Extractor, Callgraphs est implémenté en tant que greffon de GCC. Dans le cas du noyau Linux, cela nous permet de gérer certains cas délicats comme les fonctions déclarées *static* portant le même nom dans plusieurs fichiers (nous les appelons « fonctions homonymes » — rappelons que plusieurs fonctions ne peuvent pas porter le même nom en C si l'une au moins est de portée globale, ou bien si elles sont de portée *static* et déclarées dans la même unité de traduction). Tandis qu'un outil pourrait se tromper en construisant le graphe et confondre une fonction avec son homonyme, le compilateur ne peut pas commettre une telle erreur. Néanmoins, Callgraphs souffre des mêmes restrictions que tous les greffons, à savoir qu'il est exécuté de manière indépendante sur toutes les unités de traduction.

Dans les graphes d'appels, une fonction est identifiée par un nom ainsi qu'un fichier et une ligne de définition. Ces informations identifient de façon univoque une fonction en C car on ne peut pas avoir plusieurs fonctions du même nom définie dans la même portée dans ce langage. Le greffon Callgraphs s'insère au tout début de la chaîne de compilation de chaque fonction. L'algorithme de génération du graphe d'appel est très simple : on parcourt toutes les instructions de la fonction et, pour chacune d'entre elles correspondant à un appel statique de fonction (par opposition à un appel de fonction à travers un pointeur), on enregistre le fait que la fonction en cours de compilation appelle la fonction de l'instruction. À l'intérieur du greffon, on bénéficie de la connaissance de GCC et on peut donc retrouver la déclaration correspondant à la fonction. Le graphe produit est une liste d'adjacence stockée dans une base de données. Une table contient la liste des fonctions, tandis qu'une deuxième contient tous les couples ⟨ fonction appelante, fonction appelée ⟩.

Dans le cas des graphes d'appels, il faut prendre garde à un problème particulier : on peut appeler dans une unité de traduction une fonction définie dans une autre unité, à condition qu'elle ne soit pas déclarée *static* mais globale. Le problème est qu'au moment où l'on voit cet appel de fonction, on ne peut pas en voir la définition, qui se trouve dans une autre unité de traduction, mais uniquement sa *déclaration* dans l'unité de traduction courante. Par conséquent, on ne peut pas renseigner correctement le fichier et la ligne de définition. Pour pallier ce problème, nous retenons également dans la base de données la portée *static* ou globale de la fonction et nous appliquons la procédure suivante à chaque rencontre d'un appel de fonction.

- On récupère l'identifiant de la fonction courante (on l'a ajouté dans la base au début de la fonction).
- On récupère depuis l'appel la déclaration de la fonction.
- Si la déclaration montre que la fonction est *static* dans ce cas, la fonction est nécessairement définie dans la même unité de traduction.
  - Si la fonction existe déjà dans la base (même nom et même fichier) : on récupère l'identifiant dans la base.
  - Sinon : on ajoute la fonction dans la base et on récupère son nouvel identifiant.
- Sinon, la fonction est globale. Si elle est définie dans une autre unité de traduction :
  - Si la fonction existe déjà dans la base (on compare seulement le nom, car on ne peut pas avoir deux fonctions globales de même nom dans un même code) : on récupère l'identifiant dans la base.
  - Si la fonction n'existe pas déjà dans la base : on ajoute la fonction dans la base, sans renseigner son fichier ni sa ligne de définition, qui ne sont pas encore connus. On récupère le nouvel identifiant.
- Sinon, la fonction est globale et elle est définie dans l'unité de traduction courante :
  - Si la fonction existe déjà dans la base : on récupère l'identifiant dans la base. On renseigne le fichier et la ligne de déclaration qu'on peut à présent connaître.
  - Si la fonction n'existe pas déjà dans la base : on ajoute la fonction dans la base et on récupère le nouvel identifiant.
- On ajoute l'appel entre la fonction courante et cette fonction.

On remarque qu'il est impossible de produire directement le graphe d'appel au fur et à mesure que l'on parcourt le code, dans un fichier CSV par exemple. Il faut nécessairement faire deux passes sur le code ou bien utiliser un système de gestion de base de données car on ne peut pas connaître l'endroit précis de la définition de chaque fonction globale avant d'avoir analysé l'intégralité des unités de traduction. Pour limiter les dépendances de notre greffon, nous avons choisi le système de gestion de base de données SQLite [41], qui est très léger et s'appuie sur un seul fichier sur le disque, au détriment bien sûr des performances en comparaison de systèmes plus lourds à installer et administrer.

Une fois la base construite, on peut l'analyser directement ou bien en l'important dans un outil plus adapté tel que le système de gestion de base de données orientée graphe Neo4J [60]. Pour le noyau 4.7, compilé pour l'architecture x86 avec une configuration par défaut, on obtient un graphe de presque soixante-dix mille nœuds et deux cent trente mille arcs. Le système de gestion de base de données Neo4J est optimisé en particulier pour répondre à des questions comme « quels sont les crochets LSM atteignables depuis cet appel système ? », « quels sont les crochets communs à ces deux fonctions ? », etc.

4.3 Conclusion

Les outils Kayrebt::Extractor et Kayrebt::Viewer ont été présentés à la conférence VISSOFT 2015 [37]. Cette conférence a pour objet la visualisation de logiciel dans le but

de faciliter sa compréhension, son développement ou encore son débogage. Kayrebt a reçu un accueil très favorable, en particulier car il illustre une approche novatrice : l'exploitation du compilateur et des artefacts de compilation pour la production de visualisations logicielles.

En plus des bonnes propriétés que nous avons déjà décrites plus haut (capacité à traiter une base de code massive et complexe telle que le noyau Linux, interprétation du code conforme à celle du compilateur « officiel », réutilisation pour n'importe quelle architecture, configuration et version de la base de code), les outils Extractor et Callgraphs sont également relativement rapides bien que ce ne soit pas un objectif auquel nous avons attribué une grande importance. Nous avons expérimenté sur une machine dotée d'un processeur à $2.10\,\mathrm{GHz}$ et de $4\,\mathrm{Gio}$ de mémoire vive sur laquelle la compilation du noyau prend une vingtaine de minutes en temps normal. L'extraction des graphes de flot de contrôle (une compilation complète du noyau avec le greffon Kayrebt::Extractor activé) prend environ trente minutes. La production du graphe d'appel est nettement plus longue, un peu plus d'une heure, mais cela est essentiellement dû à l'usage d'une base de donnée SQLite que nous poussons au-delà de son usage nominal. Utiliser une base de données plus performante ou produire le graphe directement dans Neo4J diminuerait certainement le temps de compilation, au prix de dépendances supplémentaires.

Dans le cadre de cette thèse, les outils Kayrebt nous ont servi à appréhender à la fois le code du noyau et le compilateur GCC. En effet, les graphes permettent de saisir d'un coup d'œil la taille d'une fonction et les chemins d'exécutions. En particulier, cela permet de saisir d'un seul coup d'œil les chemins qui atteignent un certain point d'une fonction. Comme le greffon Extractor peut être placé à des endroits variables de la chaîne de compilation, les graphes peuvent montrer différentes représentations intermédiaires du code construites par GCC, ce qui permet d'apprécier la façon dont GCC transforme le code durant les phases d'optimisations.

Chapitre 5

Vérification du placement des crochets LSM pour le contrôle de flux d'information

LSM a été introduit dans le noyau en 2001 [104, 105], dans l'objectif de fournir une plate-forme pour intégrer des modules de sécurité dans le noyau qui soit totalement indépendante du fonctionnement et des objectifs des modules en eux-mêmes. En d'autres termes, les développeurs du noyau ne voulaient pas forcer la main aux distributions Linux et aux utilisateurs quant au choix de leur module de sécurité mais souhaitait malgré tout fournir aux développeurs une interface unique et maintenable pour s'intégrer plus facilement dans le noyau. Cette initiative s'est révélée globalement payante bien qu'elle ait ses détracteurs, comme le groupe grsecurity [87] ou RSBAC [68]. Ces projets critiquent notamment le fait que réimplémenter leur projet (antérieur à LSM) coûterait beaucoup d'efforts ; que LSM semble n'avoir été conçu que pour le contrôle d'accès en général, et SELinux [85] en particulier ; et enfin qu'il est presque impossible d'avoir deux modules LSM actifs en même temps. Certaines critiques sont caduques aujourd'hui car l'interface a beaucoup évolué. D'autres modules que SELinux ont été intégrés avec succès dans les sources officielles comme Smack en février 2008 [97, commit e114e473771c848c3cfec05f0123e70f1cdbdc99], Tomoyo en février 2009 [97, commit c73bd6d473ceb5d643d3afd7e75b7dc2e6918558], Apparmor en juillet 2010 [97, commit cdff264264254e0fabc8107a33f3bb75a95e981f]. De plus, depuis la version 4.2, il est possible d'avoir plusieurs modules LSM simultanément, indépendamment du comportement des modules eux-mêmes [97, commit e22619a29fcdb513b7bc020e84225bb3b5914259]. Si plusieurs modules sont présents, ils sont simplement tous consultés lors de chaque décision de sécurité. Cela a d'ailleurs rendu possible l'apparition de « petits » modules, avec une mission unique, comme LoadPin qui vérifie que tous les fichiers chargés par le noyau (comme les modules et le firmware) proviennent d'un même système de fichiers.

L'un de ces sujets de préoccupations reste néanmoins actuel : le fait que LSM a été principalement conçu dans l'objectif du contrôle d'accès et que donc rien ne garantit qu'il soit approprié à d'autres objectifs, en particulier le contrôle de flux d'information. Cependant, plusieurs modules de sécurité comme Laminar [76], KBlare [33] et Weir [67] sont implémentés avec LSM sans qu'aucune de ces approches ne discutent de l'adéquation de ce *framework* avec leurs objectifs ; la question est donc de première importance.

Dans ce chapitre, après un descriptif de la conception et de l'implémentation de LSM, nous posons le problème de l'utilisabilité de LSM pour l'implémentation du contrôle de flux d'information et nous proposons notre approche pour y répondre. Nous concluons que LSM est effectivement approprié pour implémenter un mécanisme de contrôle de flux d'information, non cependant sans quelques adaptations nécessaires dues à la différence entre contrôle d'accès et contrôle de flux.

5.1 LSM et les moniteurs de flux d'information

LSM fournit aux développeurs de modules de sécurité deux éléments :

— des champs supplémentaires dans un certain nombre de structures de données internes du noyau, intentionnellement inutilisés dans le code du noyau et laissés à la disposition des modules de sécurité pour stocker l'état nécessaire aux décisions de sécurité à prendre ;

— des points dans le code du noyau, appelés crochets, où peuvent être enregistrés par les modules de sécurité des fonctions.

Les crochets forment une interface assez vaste. Dans la version 4.7 du noyau, on en dénombre cent quatre-vingt-dix-huit, répertoriés en annexe A. À chaque crochet, un module de sécurité est libre d'associer où non une fonction. À chaque fois qu'un *thread* rencontre un crochet au cours de son exécution dans le noyau, les fonctions attachées à ce crochet sont exécutées dans l'ordre de leur enregistrement, quelque soit le module qui les a enregistrées. Si l'une d'entre elles retourne un code d'erreur, cela court-circuite les fonctions enregistrées suivantes, qui aurait normalement dû être appelées à la suite.

On peut distinguer deux types de crochets. Certains sont prévus afin de permettre aux modules de sécurité de maintenir leur état. Il s'agit des crochets permettant notamment d'allouer et désallouer les champs de sécurité dans les structures, ou encore prévenant les modules qu'une certaine opération est terminée afin de permettre la mise à jour de ces champs de sécurité. Ces crochets n'ont typiquement pas de valeur de retour. L'autre type de crochets est celui des crochets d'autorisation. Ces crochets sont placés dans le code des appels système, avant les opérations considérées comme sensibles, que les modules de sécurité pourraient désirer interdire sélectivement. Ces crochets fournissent une valeur de retour sous la forme habituelle d'un 0 si tout s'est bien passé et d'un des codes d'erreur spécifié par POSIX sinon. Les modules de sécurité renvoient généralement l'erreur `EPERM`, signifiant « permissions insuffisantes ». Ces erreurs sont propagées dans la fonction appelante du noyau, puis au code de l'espace utilisateur ayant demandé l'appel système. De la sorte, le mécanisme d'interposition est parfaitement intégré au noyau. Du point de vue du processus utilisateur, il n'y a pas de différence entre un appel système ayant échoué à cause d'une erreur commune — comme des paramètres invalides, un manque de ressources ou encore des permissions insuffisantes détectées par le contrôle d'accès discrétionnaire du noyau — et une erreur rapportée par un des modules de sécurité.

L'une des caractéristiques peut-être surprenante du framework LSM est l'absence de mécanisme permettant d'outre-passer les décisions prises par le noyau. En effet, le noyau applique un certain nombre de contrôles à chaque appel système. Ces contrôles sont en général absolument nécessaires, comme vérifier qu'un appel système `read` s'effectue bien sur un descripteur de fichier valide et ouvert par le processus appelant. D'autres contrôles relèvent du contrôle d'accès discrétionnaire traditionnel de Linux, hérité d'UNIX. Ces contrôles-ci pourraient en théorie être outre-passés par un module

LSM sans risque pour la sécurité ; néanmoins, la décision a été prise de ne pas permettre cela. Les modules de sécurité implémentés avec LSM peuvent donc uniquement appliquer plus de restrictions, et jamais en lever.

5.1.1 Appels systèmes provoquant des flux d'information

Le noyau que nous analysons est le noyau *vanilla* dans sa version 4.7. Nous avons identifié trois cent quatorze appels système mais tous ne sont pas responsables de flux d'information. En comparant la liste des appels système disponibles dans le noyau 4.7 à la liste dressée par Hauser dans sa thèse de doctorat [38] portant sur le noyau 3.2, on peut observer que pas moins de dix-sept nouveaux appels système sont apparus. L'interface des appels système maintient en permanence la rétro-compatibilité — il est hors de question de « casser l'espace utilisateur » [95] — mais elle est loin d'être figée. De nouveaux appels système apparaissent fréquemment, à l'usage des applications ou des bibliothèques système avec des besoins spécifiques.

Pour établir la liste des appels système provoquant des flux d'information (tableau 5.1), nous avons tout d'abord établi une liste exhaustive des conteneurs d'information à considérer :

— l'espace d'adressage des processus ;
— les fichiers réguliers ;
— les tubes ;
— les files de messages System V ;
— les files de messages POSIX ;
— les sockets UNIX.

On peut constater plusieurs parti-pris atypiques dans cette liste. En premier lieu, nous ne considérons pas les processus eux-mêmes comme des conteneurs comme c'est usuel dans les mécanismes de contrôle de flux mais leur espace d'adressage. En effet, c'est bel et bien la mémoire des processus qui contient des données, c'est donc elle qui constitue un conteneur d'information, et non les processus. De plus, il faut noter que les processus sont une abstraction manipulée indirectement par le noyau car, comme on l'a vu dans le chapitre 3, celui-ci ordonnance des tâches qui sont plus proches des *threads* que des processus. Ces derniers sont vus simplement comme des groupes de tâches partageant le même gestionnaire de signaux. La distinction est importante car, sous Linux, il est possible à deux *threads* de deux processus différents de partager la même mémoire (en revanche, deux *threads* d'un même processus *ne peuvent pas* avoir une mémoire différente) ; et il est également possible à deux *threads* partageant la même mémoire de la « dé-partager » (en demandant au noyau de la dupliquer). Si deux *threads* partagent la même mémoire, il est naturel de considérer qu'ils doivent également partager le label de sécurité indiquant les flux dont ils ont été les destinataires. Par conséquent, il est plus simple et plus sûr d'attacher les labels aux mémoires des *threads* plutôt qu'aux *threads* eux-mêmes.

Deuxièmement, nous ne suivons pas les flux en provenance ou à destination de l'extérieur du système. De nombreuses approches de contrôle de flux d'information empêchent, dans leur politique, que les interfaces externes deviennent taguées, ce qui prévient toute fuite d'information confidentielle vers l'extérieur du système. Il nous serait possible d'adopter une approche similaire. Cependant, cela est hors du cadre de nos travaux, qui ne portent que sur le *suivi* de flux, et non sur ce que ce suivi permet en termes d'objectifs de sécurité. Hauser a développé pour Blare [38] un moyen de

TABLE 5.1 – Flux causés par les appels système de Linux v 4.7

Appel système	Flux
Flux discrets	
`read`	Fichier → mémoire du processus appelant
`readv`	Fichier → mémoire du processus appelant
`preadv`	Fichier → mémoire du processus appelant
`pread64`	Fichier → mémoire du processus appelant
`write`	Mémoire du processus appelant → fichier
`writev`	Mémoire du processus appelant → fichier
`pwritev`	Mémoire du processus appelant → fichier
`pwrite64`	Mémoire du processus appelant → fichier
`sendfile`	Fichier → fichier
`sendfile64`	Fichier → fichier
`splice`	Fichier → tube
	Tube → fichier
	Tube → tube
`tee`	Tube → tube
`vmsplice`	Mémoire du processus appelant → tube
	Tube → mémoire du processus appelant
`recv`	Socket → mémoire du processus appelant
`recvmsg`	Socket → mémoire du processus appelant
`recvmmsg`	Socket → mémoire du processus appelant
`recvfrom`	Mémoire du processus appelant → socket
`send`	Mémoire du processus appelant → socket
`sendmsg`	Mémoire du processus appelant → socket
`sendmmsg`	Mémoire du processus appelant → socket
`sendto`	Mémoire du processus appelant → socket
`process_vm_readv`	Mémoire d'un autre processus → mémoire du processus appelant
`process_vm_writev`	Mémoire du processus appelant → mémoire d'un autre processus
`migrate_pages`	Mémoire d'un autre processus → mémoire du processus appelant
`move_pages`	Mémoire d'un autre processus → mémoire du processus appelant
`fork`	Mémoire du processus appelant → mémoire d'un nouveau processus
`vfork`	Mémoire du processus appelant → mémoire d'un nouveau processus
`clone`	Mémoire du processus appelant → mémoire d'un nouveau processus

Suite de la TABLE 5.1. Flux causés par les appels système de Linux v 4.7

`execve`	Fichier régulier exécutable → mémoire du processus courant
`execveat`	Fichier régulier exécutable → mémoire du processus courant
`msgrcv`	File de messages System V → mémoire du processus courant
`msgsnd`	Mémoire du processus courant → file de messages System V
`mq_timedreceive`	File de messages POSIX → mémoire du processus courant
`mq_timedsend`	Mémoire du processus courant → file de messages POSIX
Flux continus	
`shmat`	Mémoire partagée POSIX ↔ mémoire du processus courant
`mmap_pgoff`	Fichier régulier ou périphérique ↔ mémoire du processus courant
	Fichier régulier ou périphérique → mémoire du processus courant
`mmap`	Fichier régulier ou périphérique ↔ mémoire du processus courant
	Fichier régulier ou périphérique → mémoire du processus courant
`ptrace`	Mémoire du processus courant ↔ mémoire d'un autre processus

suivre les teintes à l'intérieur d'un réseau contrôlé en transmettant des teintes grâce à un marquage des paquets IP. Nous n'avons pas repris cette idée car nos travaux de recherche se bornent à l'échelle du système d'exploitation, mais ce serait une évolution envisageable.

5.1.2 LSM pour le contrôle de flux d'information

Tous les moniteurs de flux d'information implémentés pour Linux que nous avons étudiés utilisent les crochets LSM pour implémenter le suivi de flux d'information : Laminar [76], KBlare [33, 40], AndroBlare [2], Flume [51], Weir [67]. On peut ajouter à cette liste Flowx [16, 17] qui utilise les crochets essentiellement pour poly-instancier des conteneurs d'information selon leur niveau de sécurité. Il ne s'agit pas vraiment de contrôle de flux d'information selon notre définition mais plutôt du contrôle d'accès obligatoire, bien que la délimitation devienne un peu floue à ce point. En effet, Flowx sépare les objets selon leur niveau de sécurité plutôt que d'autoriser et interdire les flux au cas par cas en fonction des flux passés. TaintDroid [28] utilise également les crochets pour sauvegarder les labels des fichiers dans les attributs étendus du système de fichiers, tout en appliquant le contrôle de flux d'information à un niveau plus fin. Tous ces moniteurs font donc l'hypothèse, plus ou moins explicitement, qu'en interceptant toutes les exécutions passant par un crochet LSM, ils sont capables d'observer l'intégralité des flux d'information (du moins, sur les canaux qu'ils affirment

couvrir). À notre connaissance, aucun travail n'a jusqu'alors porté sur la vérification précise de cette hypothèse, pourtant cruciale. Des travaux connexes, discutés dans le chapitre 2, section 2.5.4, ont vérifié le positionnement des crochets pour l'accès aux structures de données internes du noyau, mais elles laissent des doutes quant à notre cas d'usage de LSM, le suivi de flux. L'analyse du code source de Laminar [75] et KBlare [39] offre cependant des éléments de réponse. Elle montre que l'ajout de crochets LSM supplémentaires s'est révélé nécessaire pour effectuer le suivi de flux d'information dans ces deux outils. Laminar ajoute au total huit nouveaux crochets, pour allouer et manipuler les labels et les capacités, ainsi qu'un crochet permettant d'autoriser ou non un processus à prendre connaissance que le tube dans lequel il tente d'écrire est plein. Ces crochets permettent l'implémentation de son modèle ainsi que la fermeture du canal caché constitué par les tubes. KBlare a ajouté un crochet permettant de surveiller le détachement des mémoires partagées System V pour savoir quand le flux continu entre un processus et une mémoire partagée prend fin. Weir pour sa part n'a ajouté aucun crochet mais couvre moins d'IPC UNIX « classiques » que les deux premiers. Il couvre en revanche le binder d'Android qui est un bus central et une des principales sources de flux dans les systèmes Android. Dans une autre approche notable, Flowx, qui n'implémente pas cependant selon notre définition de contrôle de flux d'information, les auteurs ont dû ajouter de nombreux crochets. Ces nouveaux crochets permettent la poly-instanciation de tous les conteneurs d'information (processus, fichiers, files de messages, etc.) afin de garantir une propriété très forte de non-interférence entre les processus de différents niveaux de sécurité. Bien que l'ajout de ces crochets ne soit pas fait dans l'optique de propager des teintes, il faut cependant noter que ces crochets sont nécessaires car Flowx cherche à appliquer des politiques de flux, plus qu'à protéger les structures de données du noyau. En conséquence, les crochets ajoutés par Flowx, tout comme ceux proposés par Laminar et KBlare, indiquent que LSM est globalement adapté pour le suivi de flux d'information — puisque, de fait, il est employé par de nombreuses approches — mais partiellement déficient sur certains points.

5.2 Décider du bon positionnement des crochets

5.2.1 Conception d'un appel système

Un appel système sous Linux est codé de la manière présentée dans l'extrait 5.1. La déclaration commence avec une macro `SYSCALL_DEFINE` comprenant le nom de l'appel système et ses arguments. Ce type de déclaration, inhabituel, est nécessaire pour générer à la compilation une fonction dans le noyau propre à être appelée depuis l'espace utilisateur, ce qui implique des vérifications statiques et des conversions de types particulières à l'entrée et au retour de la fonction. Certains paramètres de l'appel sont marqués de l'attribut `__user` indiquant qu'ils sont des pointeurs originaires de l'espace utilisateur, donc faisant référence à l'adressage virtuel utilisé par le processus appelant. Cette information disparaît à la compilation mais peut être utilisée par des analyses statiques pour vérifier qu'aucun pointeur marqué `__user` n'est déréférencé directement par le noyau sans vérifier sa validité. Dans le contexte du suivi de flux d'information, la présence d'un pointeur `__user` est une heuristique intéressante pour supposer que l'appel système cause un flux de la mémoire du processus appelant vers d'autres conteneurs d'information.

Un appel système est normalement composé de trois parties :

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf,
    size_t, count)
{
  struct fd f = fdget_pos(fd);
  ssize_t ret = -EBADF;

  if (f.file) {
    loff_t pos = file_pos_read(f.file);
    ret = vfs_read(f.file, buf, count, &pos);
    if (ret >= 0)
      file_pos_write(f.file, pos);
    fdput_pos(f);
  }
  return ret;
}
```

EXTRAIT 5.1 – Implémentation de l'appel système `read`

```
int rw_verify_area(int read_write, struct file *file, const
    loff_t *ppos, size_t count)
{
  struct inode *inode;
  loff_t pos;
  int retval = -EINVAL;

  inode = file_inode(file);
  if (unlikely((ssize_t) count < 0))
    return retval;
  pos = *ppos;
  if (unlikely(pos < 0)) {
    if (!unsigned_offsets(file))
      return retval;
    if (count >= -pos) /* both values are in 0..LLONG_MAX */
      return -EOVERFLOW;
  } else if (unlikely((loff_t) (pos + count) < 0)) {
    if (!unsigned_offsets(file))
      return retval;
  }

  if (unlikely(inode->i_flctx && mandatory_lock(inode))) {
    retval = locks_mandatory_area(inode, file, pos,
      pos + count - 1, read_write == READ ? F_RDLCK :
     F_WRLCK);
    if (retval < 0)
      return retval;
  }
  return security_file_permission(file,
        read_write == READ ? MAY_READ : MAY_WRITE);
}
```

EXTRAIT 5.2 – Fonction `rw_verify_area`

```
ssize_t __vfs_read(struct file *file, char __user *buf,
    size_t count, loff_t *pos)
{
  if (file->f_op->read)
    return file->f_op->read(file, buf, count, pos);
  else if (file->f_op->read_iter)
    return new_sync_read(file, buf, count, pos);
  else
    return -EINVAL;
}
```

Extrait 5.3 – Fonction `__vfs_read`

- La première partie consiste à analyser les arguments de l'appel système et vérifier leur validité, puis à récupérer les structures de données nécessaires à l'opération de l'appel système. Il est généralement nécessaire aux processus de synchroniser les accès aux structures de données avec les autres processus s'exécutant en parallèle, en utilisant un des mécanismes de verrouillage du noyau. Les vérifications les plus basiques sont faites en premier, suivies des vérifications plus poussées, y compris celles des modules de sécurité LSM. Les appels système peuvent appeler de nombreuses fonctions imbriquées, comme `read` qui appelle `vfs_read` qui elle-même appelle `rw_verify_area` et `__vfs_read`. La fonction `rw_verify_area` (présentée dans l'extrait de code 5.2) vérifie que le fichier peut être lu par le processus (il est impossible de lire un fichier verrouillé ou de lire au-delà de la fin du fichier) et appelle le crochet LSM `security_file_permission`. Ce crochet est normalement utilisé par les modules LSM pour implémenter des vérifications supplémentaires et empêcher la lecture de fichiers sous certaines conditions. Si une vérification échoue, il est nécessaire de déverrouiller les structures et défaire les opérations entamées dans l'ordre inverse de l'ordre de verrouillage.
- La deuxième partie est l'opération de l'appel système proprement dite. Dans le cas de l'appel système `read`, il s'agit de la copie des informations du fichier lu vers le tampon mémoire passé par le processus. L'opération de lecture est faite par la fonction `__vfs_read` détaillée dans l'extrait 5.3. Dans le cas des appels système manipulant des fichiers, réaliser l'opération demandée revient généralement à appeler une fonction dépendante du système de fichiers, via un pointeur de fonction dans la structure de données représentant le fichier. Dans le cas de `read`, le noyau appelle une fonction enregistrée dans le pointeur `read` dans la structure `f_op` de type `struct file_operations` du fichier.
- Enfin, la troisième partie consiste à défaire ce qui est fait dans la première (comme déverrouiller les structures de données verrouillées) ainsi qu'à enregistrer certaines statistiques, collecter des traces d'accès aux structures, etc. Enfin, l'appel système doit retourner un entier. Cet entier est toujours zéro si l'appel s'est déroulé correctement, ou un code de retour négatif dont la valeur absolue est spécifiée par POSIX. Par exemple, `EBADF` est la valeur retournée par un appel système lorsqu'un descripteur de fichier invalide lui est passé en paramètre.

Les modules de sécurité Linux peuvent uniquement prendre une décision de sécurité lorsque l'exécution d'un *thread* atteint un crochet LSM car ce sont à ces endroits uniquement que le module de sécurité peut attacher une fonction. En d'autres termes,

les modules de sécurité implémentés avec le *framework* LSM ne peuvent pas observer l'intégralité de l'exécution des appels système mais uniquement certains points prédéfinis. Il n'est donc possible de détecter un flux et de faire évoluer l'état du système pour enregistrer l'occurrence de ce flux que dans une fonction attachée à un crochet. Par conséquent, la position des crochets est cruciale pour le suivi correct des flux. En étudiant les appels système générant des flux d'information et la liste des crochets et leur position, il est possible de savoir à quels crochets il convient d'attacher une fonction implémentant le suivi de flux. De manière équivalente, s'il existe un appel système générant un flux d'information et qu'il est possible d'effectuer le flux sans passer par un crochet LSM avant, alors il existe un moyen d'esquiver le suivi de flux, et donc par exemple les politiques de sécurité mises en place. Une condition nécessaire à un suivi de flux correct est donc la présence d'un crochet LSM avant chaque flux dans les appels système en générant. La présence d'un crochet après l'exécution d'un flux est moins intéressante ici car il ne permettrait pas d'*empêcher* une opération illégale en cours. Néanmoins, nous verrons au chapitre suivant que les crochets post-flux sont nécessaires pour une autre raison : la gestion de la concurrence entre flux.

5.2.2 Problèmes spécifiques au contrôle de flux d'information

Nos recherches préliminaires nous ont conduit à identifier plusieurs différences en ce qui concerne le placement des crochets entre le contrôle d'accès et le contrôle de flux d'information.

En premier lieu, les appels système qui doivent être surveillés par le contrôle d'accès sont ceux qui créent une ressource retournée au processus utilisateur appelant, et qui lui donne un moyen d'accéder à un conteneur d'information. Par exemple, l'appel système `open` crée un descripteur de fichier. Ce descripteur de fichier peut ensuite servir au processus pour demander les appels système `read` et `write`. Dans un sens, le descripteur de fichier sert de jeton d'autorisation permettant au processus d'accéder au contenu du fichier, le contrôle est fait avant de délivrer ce jeton. Parfois, cependant, il est nécessaire de révoquer ce jeton. Imaginons le scénario suivant : un utilisateur A permet à un autre utilisateur B d'accéder à un certain fichier. Le processus de B fait l'appel système `open` et se voit retourner un descripteur de fichier, avec lequel il commence à lire le fichier. Par la suite, A se ravise et interdit l'accès du fichier à B. Tant que le processus de B ne ferme pas le descripteur de fichier, il peut continuer à l'utiliser pour lire le fichier. Pour cette raison, un crochet LSM est également présent dans `read` et `write` afin de revalider l'accès au fichier lors de ces opérations. La revalidation consiste à vérifier que les conditions réunies lors de l'appel système `open` sont toujours d'actualité. Le contrôle de flux d'information, au contraire du contrôle d'accès, nécessite de surveiller les appels système provoquant les flux comme `read` et `write` et ignore les appels système comme `open` qui manipulent les structures de données du noyau sans impacter les conteneurs d'information. On peut donc conclure que seuls les crochets de revalidation sont d'intérêt pour le contrôle de flux d'information.

Nous avons fait l'hypothèse que tous les flux nécessitent un appel système ; cependant, certains appels système peuvent causer des flux se produisant *après* la fin de l'appel système. Nous distinguons les flux *discrets*, générés entièrement au cours d'un appel système, des flux *continus* qui sont rendus possibles par un appel système et terminés par un autre. Un exemple de la première catégorie est le flux créé par l'appel système `read`. Lorsque l'appel système se termine, le flux du fichier vers la mémoire du processus est terminé (ou bien n'a pas eu lieu et l'appel système a retourné une erreur). La deuxième catégorie comprend les flux en mémoire. L'appel système `mmap` est

utilisé pour projeter un fichier en mémoire. Cette opération consiste à incorporer dans l'espace d'adressage du processus les pages formant le cache du fichier (c'est-à-dire les pages depuis lesquelles le fichier est lu et écrit et qui sont synchronisées sur le support physique du fichier de temps en temps). Une fois le fichier projeté, il est possible pour le processus de lire (et modifier, selon les permissions de la zone mémoire) le contenu du fichier en utilisant uniquement des lectures–écritures en mémoire, qui ne réclament pas d'appels système. Un autre appel système, `munmap` peut défaire la projection en mémoire d'un fichier. Par conséquent, on peut dire qu'il y a un flux d'information continu entre le processus et le fichier démarré par `mmap` et terminé par `munmap`. Dans le cas du contrôle d'accès, il est suffisant de vérifier la projection et il serait illogique de permettre `mmap` puis d'interdire `munmap`, donc aucun crochet n'est nécessaire dans `munmap`. Dans le cas du contrôle de flux, la situation est différente car, comme on l'a vu, l'appel système `munmap` permet de connaître la fin du flux continu. Sans crochet dans cet appel, le suivi de flux est impossible (sauf à pouvoir connaître la fin du flux par un autre moyen, ce qui est en réalité l'approche que nous avons retenue dans le chapitre 6).

5.3 Analyse statique vérifiant la bonne position des crochets

La propriété essentielle que nous voulons vérifier est la présence d'un crochet LSM avant chaque instruction causant un flux dans les appels système concernés. Nous avons appelé cette propriété la *médiation complète*.

5.3.1 Position des crochets LSM et des instructions causant les flux

La liste des appels système générant des flux, présentée dans le tableau 5.1, a été établie en nous basant sur un précédent travail de Hauser [38] portant sur l'implémentation du moniteur de flux d'information *KBlare*. La liste établie par Hauser a été mise à jour pour correspondre au noyau 4.7. Il est important de noter que la non-exhausivité de cette liste n'invalide pas notre approche car elle ne constitue qu'un paramètre de notre analyse, chaque appel système étant analysé indépendamment.

La position des crochets LSM peut être extraite automatiquement en étudiant le graphe d'appel de fonctions des appels système. Nous avons utilisé pour ce faire Kayrebt::Callgraphs. Ce greffon du compilateur GCC présenté au chapitre 4 permet de calculer et d'exporter, pendant la compilation, le graphe d'appels de chaque fonction, afin de permettre de les étudier avec des outils adaptés ; dans notre cas, une base de données orientée graphes, Neo4j. La détermination des instructions constituant le flux d'information proprement dit est beaucoup plus floue en général. En effet, contrairement aux crochets LSM, on se heurte ici à plusieurs problèmes :

— Effectuer un flux n'est pas, en général, une opération atomique. Il n'y a donc pas une seule instruction mais toute une branche du code qui génère le flux. Dans ce cas, il est naturel de considérer la première instruction de la branche comme point de départ du flux, et de rechercher un crochet LSM avant ce point.

— Il n'est pas toujours aisé de dire à quel point le flux est effectué réellement. Lorsque l'on envoie un message sur une *socket* réseau, faut-il considérer que le flux est effectué lorsque le paquet de données a été copié dans la mémoire du

pilote de la carte réseau, lorsque le message est envoyé, lorsqu'il est acquitté par le destinataire ? L'écriture dans un fichier local soulève le même type de questions : une écriture est-elle faite lorsque le cache du fichier est modifié, lorsque la taille du fichier est changée, ou bien lorsque les modifications sont synchronisées sur le disque ?

— Un flux peut prendre de nombreuses formes. Il s'agit de modifications de zones de mémoire correspondant à des conteneurs d'information comme le cache pour les fichiers, des tampons de mémoires pour les sockets réseau, etc. Ces modifications peuvent être faites via un appel à une fonction telle que `memcpy`, ou bien une affectation directe, ou bien encore la modification de l'organisation de la mémoire de processus utilisateur.

Une analyse manuelle nous permet d'identifier les points de génération des flux dans les appels système. Nous considérons qu'un flux est effectué lorsque ses effets sont visibles par les autres processus. Par exemple, quand un processus modifie les pages de mémoire associées à un fichier dans le cache, les autres processus vont pouvoir lire ces nouvelles données : on doit donc considérer que le flux est effectué, même si la synchronisation du cache avec le disque n'est pas encore effective et ne sera réalisée que plus tard. D'autres heuristiques nous guident également. Premièrement, les opérations des appels système requièrent généralement de verrouiller ou d'incrémenter le compteur de référence de structures de données manipulées. On sait donc que l'opération importante de l'appel système est probablement effectuée lorsque le maximum de structures sont verrouillées. Enfin, en particulier dans le cas des appels système concernant les fichiers ou le réseau, les opérations sont déléguées au sous-module du noyau responsable du système de fichiers ou du protocole réseau utilisé. On peut donc repérer les appels à ces sous-modules pour déterminer quand est fait le flux.

5.3.2 Graphes de flot de contrôle

Notre analyse porte sur le code des appels système, que nous représentons, de manière classique, sous la forme de graphes de flot de contrôle. Plusieurs points sont à noter, qui distinguent notre approche de la plupart des autres analyses. En premier lieu, nos graphes ne représentent pas du code C, le langage de programmation du noyau, mais une représentation intermédiaire du compilateur GCC, appelée GIMPLE. En réalité, notre analyse statique est implémentée comme un greffon du compilateur GCC appelé Kayrebt::PathExaminer2 [1]. Les graphes sont semblables à ceux extraits par Kayrebt::Extractor (voir le chapitre 4) mais PathExaminer2 travaille directement depuis l'intérieur de GCC, sur la représentation interne des graphes, et non sur des graphes extraits au format Graphviz. Les raisons et avantages de cette approche sont détaillés en sous-section 5.7. La figure 5.1 donne un exemple de graphe de flot de contrôle tel que manipulé par notre outil. Il représente la fonction `vfs_read`, qui inclut la fonction `rw_verify_area` (on reconnaît dans le graphe les appels aux fonctions `locks_mandatory_area` et `security_file_permission` des lignes 22 et 27 de l'extrait 5.2).

En effet, lors de la compilation d'un code en langage C, il est fréquent que le compilateur pratique une optimisation appelée *inlining* consistant à remplacer un appel de fonction par le corps de cette dernière. Ainsi, dans le graphe de `vfs_read`, les fonctions `rw_verify_area` et `__vfs_area` sont-elles *inlinées*, ce qui fait que l'on voit à la fois le passage dans le crochet LSM et le début de la branche où le flux est

1. Il a existé une première version abandonnée depuis, d'où le nom.

généré. Pour notre analyse, nous avons forcé l'*inlining* des appels systèmes considérés, au-delà de ce qui semblait raisonnable au compilateur, de sorte à avoir les crochets LSM et les points de génération des flux dans le même graphe. Cela simplifie l'analyse mais ne change pas la sémantique du code, car l'opération reste sous le contrôle du compilateur.

Nous appelons $\mathcal{V}$ l'ensemble des nœuds d'un graphe. Nous distinguons cinq types de nœuds présents dans les graphes, que nous décrivons ici informellement (la définition précise se trouve plus loin en section 5.4) :

$$\mathcal{V} = \mathcal{V}^{assign} \uplus \mathcal{V}^{mem} \uplus \mathcal{V}^{join} \uplus \mathcal{V}^{\varphi} \uplus \mathcal{V}^{call}$$

- Les nœuds d'affectation forment le sous-ensemble $\mathcal{V}^{assign}$. Dans le graphe de la figure 5.1, ils se présentent sous la syntaxe `<ssa 182>.86 = count.14 ‘min‘ 2147478552`. La sémantique intuitive est que la variable à gauche du signe « = » reçoit la valeur de l'expression de droite.
- Les nœuds d'affectation via pointeurs sont le sous-ensemble $\mathcal{V}^{mem}$. Ils s'écrivent par exemple `*<ssa 12>13 = 0`. La variable pointée par la valeur à gauche du signe =, qui est un pointeur, reçoit la valeur de l'expression de droite.
- Les nœuds de jonction sont le sous-ensemble $\mathcal{V}^{join}$. Ils n'ont pas de contrepartie directe dans la syntaxe du langage C mais ils simplifient celle du graphe, où ils sont représentés par des losanges sans texte, et facilitent la construction de notre modèle. En effet, ce sont les seuls nœuds à pouvoir avoir plusieurs arcs entrants ou sortants. Ces nœuds correspondent au début et fin des *basic blocks*, c'est-à-dire des séquences d'instructions qui se suivent inconditionnellement. Lorsqu'un nœud de jonction a plusieurs arcs sortants, cela correspond obligatoirement à un branchement conditionnel, et les gardes des arcs sont complémentaires : ce sont des conditions sur les mêmes variables, de telle sorte qu'à toute valuation de ces variables correspond un unique arc dont la condition est vérifiée.
- Les nœuds ϕ, formant l'ensemble $\mathcal{V}^{\varphi}$, sont des nœuds d'affectation un peu particuliers. Ils suivent toujours un nœud de jonction ayant plusieurs arcs entrants et représentent le fait qu'à un certain point de programme, une variable peut avoir des valeurs différentes selon le chemin suivi pour atteindre ce point. Par exemple, dans le graphe de la figure 5.1, le nœud `<ssa 184>.88 = PHI <<ssa 183>.87, retval.83, retval.85>` est un nœud ϕ où la variable à gauche du signe « = » reçoit la valeur de l'une des trois variables en partie droite, selon la branche prise pour atteindre le nœud de jonction juste avant.
- Les nœuds d'appel de fonction forment le sous-ensemble $\mathcal{V}^{call}$. Ils représentent soit l'appel d'une fonction, et l'affectation éventuelle d'une variable de retour, soit l'exécution d'un bloc de code en assembleur. En effet, dans notre modèle, ces deux types d'instructions constituent la même sorte de « boîte noire », pouvant modifier l'état de la mémoire du programme de manière arbitraire. Même en *inlinant* certaines fonctions, il reste des appels dans nos graphes, notamment les appels dynamiques, au travers de pointeurs de fonction.

Comme au chapitre 4, le graphe est sous forme SSA (*Static Single Assignment*). Lorsqu'une variable du code source initial reçoit plusieurs valeurs successivement, elle est dupliquée par le compilateur. Lorsque ces valeurs différentes dépendent du chemin suivi, la variable résultante reçoit sa valeur dans un nœud ϕ. De nouvelles variables sont également nécessaires pour casser les expressions arithmétiques complexes et faire en sorte que les expressions en partie droite des nœuds d'affectation contiennent

au plus un seul opérateur (forme souvent appelée « code trois adresses » : les trois adresses étant la variable de retour plus les deux opérandes). Toutes ces modifications du code original sont faites par le compilateur pour faciliter ses propres optimisations et analyses statiques et nous en profitons dans la définition et implémentation de notre analyse.

5.3.3 Propriété de médiation complète

Pour vérifier la propriété de médiation complète, nous analysons chaque graphe individuellement. Chaque graphe de flot de contrôle représente exactement un appel système. Nous considérons l'ensemble $\mathcal{P}aths$ de tous les chemins du graphe. Parmi ces chemins, nous distinguons deux sous-ensembles particuliers. Le premier ensemble $\mathcal{P}aths_{flows}$ comprend tous les chemins commençant au nœud initial du graphe (l'unique nœud sans prédecesseur, correspondant à l'entrée de l'appel système), se terminant à un nœud sans successeur du graphe (correspondant donc à un retour de l'appel) et passant par le nœud correspondant à une instruction générant un flux. Ces chemins sont ceux pouvant correspondre à des exécutions de la fonction, depuis son entrée jusqu'au retour, responsables d'un flux d'information. Si un appel système comprend plusieurs points de génération d'un flux d'information, on analyse indépendamment tous ces points. Le deuxième sous-ensemble noté $\mathcal{P}aths_{LSM}$ est celui de tous les chemins de $\mathcal{P}aths_{flows}$ comprenant des nœuds correspondant à des crochets LSM situés avant le nœud générant le flux. Naturellement, la propriété de médiation complète est satisfaite trivialement si $\mathcal{P}aths_{flows} = \mathcal{P}aths_{LSM}$. Cependant, ce n'est pas le cas de tous les appels système. On note

$$\mathbf{P} = \mathcal{P}aths_{flows} \setminus \mathcal{P}aths_{LSM}$$

l'ensemble des chemins potentiellement problématiques, les chemins générant des flux inobservables par un moniteur de flux construits avec LSM.

Certains chemins de $\mathbf{P}$ sont cependant impossibles, c'est-à-dire qu'ils ne correspondent à aucune exécution réelle du programme ; par exemple parce qu'une exécution qui emprunterait ce chemin passerait à la fois par une branche conditionnelle où une certaine variable booléenne doit être vraie puis par une autre où cette même variable est fausse sans avoir été réaffectée entre temps (ce qui est vrai dans tous les cas où la variable obéit à la forme *Single Static Assignment*). On note $\mathbf{I} \subseteq \mathcal{P}aths$ l'ensemble des chemins impossibles du graphe. La figure 5.1 présente un exemple de chemin impossible. Le chemin marqué en gras fait partie de $\mathbf{P}$, il esquive le crochet LSM et passe dans la branche causant le flux. Cependant, il requiert que la variable `ret.21` ait la même valeur que `<ssa 184>.88`, elle-même ayant la même valeur que `retval.83`. Cependant, ce chemin comprend deux arcs, l'un portant la contrainte `retval.83 < 0` et l'autre `ret.21 >= 0`, qui sont donc incompatibles.

On peut donc exprimer la propriété de complète médiation comme suit :

Propriété 1 (Médiation complète)**.** La médiation complète est vérifiée si, et seulement si, $\mathbf{P} \subseteq \mathbf{I}$.

L'objectif de notre analyse est donc de tester les chemins de $\mathbf{P}$ pour prouver qu'ils sont tous impossibles. Cependant, à cause de la présence de boucles dans le code, l'ensemble $\mathbf{P}$ est généralement infini, mais il est possible de n'analyser que les chemins acycliques (qui sont en nombre fini dans un graphe fini) pour conclure sur l'ensemble des chemins. Nous détaillons ce point dans la section 5.4.5. La figure 5.2 récapitule les ensembles de chemins considérés et leur relation.

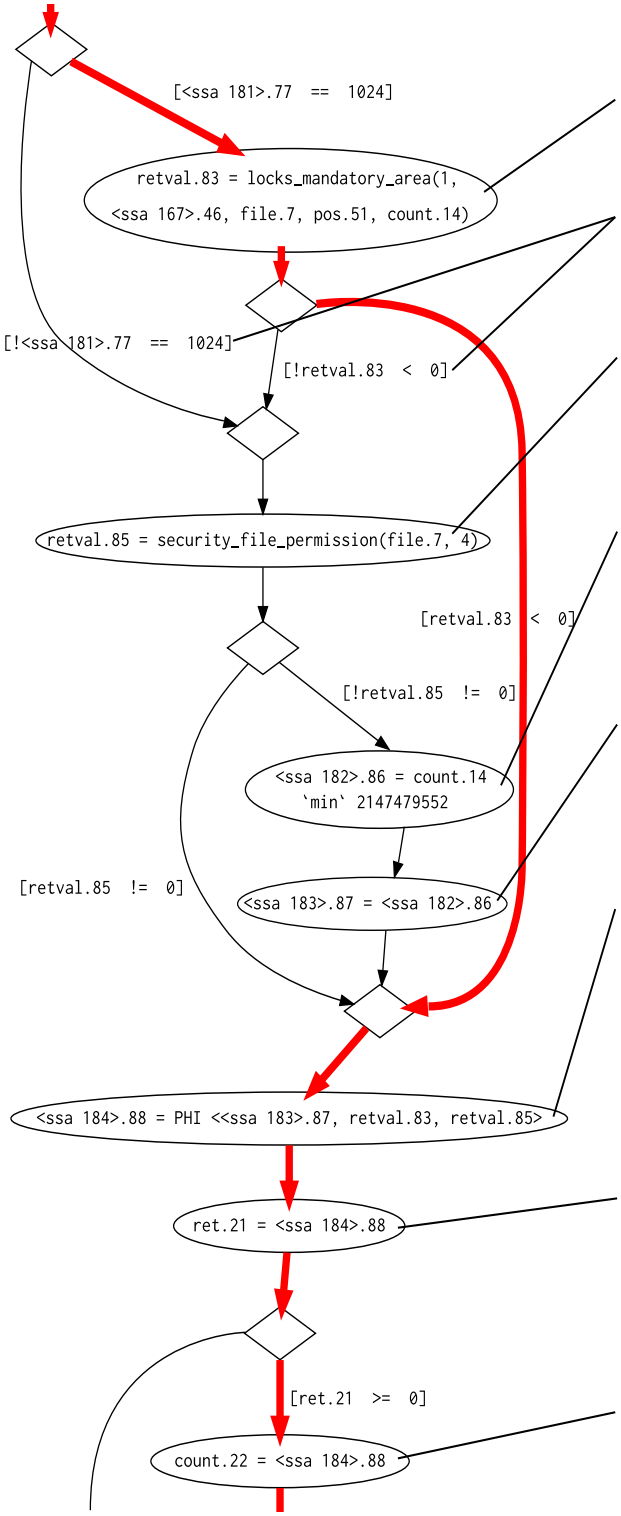


FIGURE 5.1 – Exemple de graphe de flot de contrôle — Appel système `read`

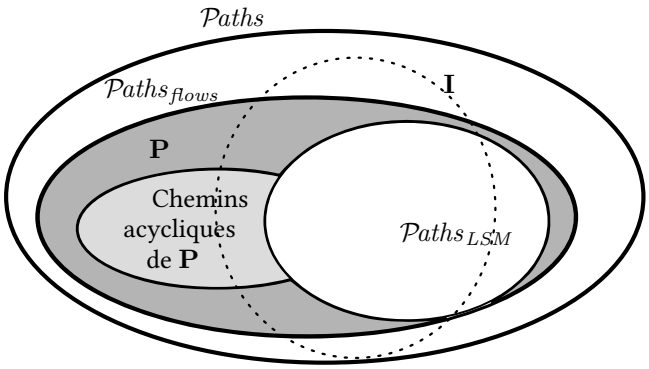


FIGURE 5.2 – Chemins étudiés dans l'analyse

5.4 Formalisation

Dans cette section, nous nous attachons à décrire formellement les objets de notre analyse, c'est-à-dire les graphes, les variables, les expressions, les instructions et leur sémantique. Notre approche est structurée de la manière suivante :

1. Les graphes de flot de contrôle forment un modèle du code, chaque graphe représentant un appel système produisant un flux d'information.
2. Les nœuds du graphe représentent les instructions et les arcs matérialisent le fait que l'exécution d'une instruction peut suivre l'exécution d'une autre.
3. Dans ces graphes, les chemins atteignant les nœuds représentant des instructions générant des flux d'information et ne passant pas par des nœuds représentant des crochets LSM sont problématiques car ils représentent un moyen d'effectuer un flux d'information que le moniteur ne peut pas détecter. Vérifier la propriété de complète médiation revient à démontrer que tous ces chemins sont impossibles.
4. Pour ce faire, nous supposons l'existence d'une sémantique concrète du code, cette sémantique est celle attribuée par GCC au code. Nous formalisons quelques hypothèses élémentaires sur cette sémantique.
5. Ensuite, nous construisons une sémantique abstraite nous permettant de discriminer les chemins impossibles des autres. Nous démontrons que cette sémantique abstraite est correcte vis-à-vis de la sémantique concrète, si cette dernière vérifie les hypothèses formalisées au point précédent. De la sorte, notre approche est générale, et pourrait s'adapter à d'autres compilateurs que GCC, car elle s'applique dès lors que les hypothèses simples que nous faisons sont vérifiées.
6. Sur la base de cette propriété de correction, nous démontrons ensuite qu'un chemin analysé et déclaré impossible ne peut pas être un chemin emprunté durant une exécution concrète de l'appel système, donc qu'il ne permet pas de violer la propriété de médiation complète.
7. Nous avons donc un moyen d'analyser tous les chemins de longueur finie mais cet ensemble est infini en cas de présence de boucles. Nous traitons ce problème en quotientant l'ensemble selon une relation d'équivalence afin de réunir dans une même classe d'équivalence tous les chemins identiques aux nombres d'itérations de leurs boucles près. La forme normale de chaque classe est l'unique chemin acyclique de cette classe.

8. Nous modifions la sémantique abstraite de sorte que si la forme normale est déclarée impossible, alors on puisse prouver que tous les chemins de la même classe d'équivalence sont eux aussi impossibles.
9. Il suffit donc d'analyser tous les chemins acycliques pour conclure sur l'impossibilité de tous les chemins. Cela est possible car dans un graphe fini, il y a un nombre fini de chemins acycliques.

Nous avons vérifié à l'aide de l'assistant de preuve Coq [93] les démonstrations de nos principaux théorèmes de correction de l'analyse statique vis-à-vis des hypothèses que nous avons faites sur la sémantique concrète du code ; en particulier parce que ces démonstrations sont de lourdes et fastidieuses analyses de cas. Nous incluons dans le corps de ce chapitre les définitions et les énoncés dans le langage de Coq. Le script de preuves lui-même est disponible en annexe B et sur la page web de Kayrebt : `https://kayrebt.gforge.inria.fr/proofs.html`.

5.4.1 Syntaxe des graphes

Après avoir donné une représentation intuitive des graphes analysés dans la section précédente et le chapitre 4, nous nous attachons ici à les décrire formellement. Les graphes de flots de contrôles sont des couples $(\mathcal{V}, \mathcal{E})$ avec $\mathcal{V}$ l'ensemble des nœuds du graphe et $\mathcal{E}$ l'ensemble des arcs. Chaque nœud représente une instruction dans le langage GIMPLE.

Définition 1 (Variables)**.** On note $\mathcal{V}ars$ l'ensemble des variables du graphe que nous considérons dans l'analyse. Les variables dont nous ne tenons pas compte forment l'ensemble $\overline{\mathcal{V}ars}$. L'ensemble $\mathcal{V}ars$ est partitionné de deux manières.

- $\mathcal{V}ars = \mathcal{V}ars^{\mathbb{Z}} \uplus \mathcal{V}ars^{ptr}$
- $\mathcal{V}ars = \mathcal{V}ars^{mem} \uplus \mathcal{V}ars^{temp}$

Les variables dans $\mathcal{V}ars^{\mathbb{Z}}$ contiennent des entiers tandis que les variables de $\mathcal{V}ars^{ptr}$ sont de type pointeur. Le compilateur les maintient strictement séparées (et génère des variables intermédiaires en cas de transtypage). En revanche, nous ne distinguons pas dans notre analyse les différents types d'entiers ou de pointeurs entre eux. Les variables dans $\mathcal{V}ars^{mem}$ sont celles vivant en mémoire, et qui peuvent être modifiées par une affectation à travers un pointeur, tandis que les variables dans $\mathcal{V}ars^{temp}$ ont une valeur assignée à leur création et le compilateur sait qu'elles ne sont pas modifiées par des effets de bords. C'est le cas des variables que le compilateur synthétise lui-même par exemple. GCC maintient cette distinction afin de pouvoir optimiser arbitrairement les variables de la seconde catégorie. Les optimisations portant sur les variables de la première catégorie sont plus délicates car elles peuvent être modifiées par des effets de bord, ou via des pointeurs calculés à partir d'adresses connues. Toutes les variables dont l'adresse est prise à un moment du programme sont dans $\mathcal{V}ars^{mem}$, ainsi que les variables dont l'adresse peut être calculée. Par exemple, posséder un pointeur vers un champ d'une structure est équivalent à posséder un pointeur vers n'importe quel autre champ de la structure.

On note que certaines variables (formant l'ensemble $\overline{\mathcal{V}ars}$) ne sont pas prises en compte. Il s'agit des variables globales et les variables volatiles, qui peuvent prendre des valeurs arbitraires indépendamment du code du programme à tout point de l'exécution. Nous excluons également les variables de type structure ou union lorsqu'elles font appel à de l'arithmétique de pointeurs (le compilateur peut casser une structure en autant de variables qu'elle contient de champs dans les cas simples). Exclure des variables réduit

la précision de notre approche car nous perdons des informations sur les chemins qui pourraient permettre de décider qu'ils sont impossibles mais maintient sa correction, c'est-à-dire qu'on ne déclare jamais impossible un chemin en réalité possible. Notre analyse omet volontairement de nombreuses variables car nous voulons faire aussi peu que possible d'hypothèses sur la sémantique des opérations du langage. En effet, il n'existe pas de formalisation de la sémantique attachée au dialecte du langage C utilisé pour programmer le noyau Linux par GCC. Or, ce langage est notoirement complexe, du fait entre autres choses de l'arithmétique de pointeurs arbitraire, et comporte de nombreux cas particuliers, où la sémantique d'une opération dépend en partie de la plate-forme matérielle. Dépendre le moins possible de la sémantique du langage source augmente donc la confiance dans les résultats de l'analyse statique et la rend plus générale. L'analyse est construite spécifiquement pour nos besoins et nos résultats, présentés en section 5.6, montrent qu'elle est suffisamment précise pour répondre à ceux-ci.

En Coq, nous définissons deux types, celui des variables prises en compte et celui des variables ignorées. Nous définissons trois propriétés décidables sur les variables : le fait d'être distingable, d'être ou non un pointeur et d'être ou non adressable.

```
Variable Vars : Type.
Variable IgnoredVars : Type.
Hypothesis Vars_eq_dec :∀x y:Vars, {x = y}+{x ≠ y}.

Variable pointer : Vars → Prop.
Variable addressable : Vars → Prop.
Hypothesis Addressable_dec :∀x:Vars, {addressable x}+{¬ addressable x}.
Hypothesis Pointer_dec :∀x:Vars, {pointer x}+{¬ pointer x}.
```

GCC possède un oracle d'alias noté $\mathcal{O}$ donnant pour un pointeur l'ensemble des variables sur lequel il peut possiblement pointer (cette information est déterminée par le compilateur essentiellement d'après le type des variables).

Définition 2 (Oracle d'alias)**.** L'oracle d'alias est une fonction :

$$\mathcal{O} : \mathcal{V}ars^{ptr} \to (\mathcal{V}ars^{ptr} \cap \mathcal{V}ars^{mem}) \uplus (\mathcal{V}ars^{\mathbb{Z}} \cap \mathcal{V}ars^{mem})$$

Tout pointeur peut être associé à un ensemble de variables sur lequel il est susceptible de pointer. Dans le pire des cas, cet ensemble est l'ensemble $\mathcal{V}ars^{mem}$ tout entier. L'ensemble résultat est partitionné : un pointeur peut pointer soit sur une variable numérique soit sur un pointeur mais pas les deux. De plus, la sortie est nécessairement une variable de $\mathcal{V}ars^{mem}$, c'est-à-dire une variable *aliasable*. La sortie de $\mathcal{O}$ pourrait bien sûr également contenir des variables de $\overline{\mathcal{V}ars}$ mais nous considérons la sortie privée de l'ensemble $\overline{\mathcal{V}ars}$ puisque nous ignorons ces variables dans l'analyse.

En Coq, nous définissons le pointeur comme une relation entre pointeurs et variables adressables et nous supposons qu'il est toujours possible de décider si un pointeur donné peut pointer sur une variable donnée ainsi que de savoir si le pointeur pointe sur des entiers ou bien sur des pointeurs.

```
Variable ptoracle :∀(x:Vars) (Hptr: pointer x) (a:Vars) (Haddr: addressable a), Prop.
Hypothesis pt_or_not_pt :∀p Hptr x Haddr, {ptoracle p Hptr x Haddr}+{¬ ptoracle p
    Hptr x Haddr}.
```

```
Lemma may_point_to p Hptr x:
 {exists (Hx: addressable x), ptoracle p Hptr x Hx}+
 {¬ addressable x}+
 {forall (Hx: addressable x), ¬ ptoracle p Hptr x Hx}.

Definition pointers_to_pointers p Hptr : Prop :=
 ∀q Haddr, ptoracle p Hptr q Haddr → pointer q.

Definition pointers_to_non_pointers p Hptr : Prop :=
 ∀q Haddr, ptoracle p Hptr q Haddr → ¬ pointer q.

Hypothesis cannot_point_to_both :
 ∀v Hptr, {pointers_to_pointers v Hptr}+{pointers_to_non_pointers v Hptr}.
```

Définition 3 (Expressions et valeurs)**.** Nous distinguons cinq types d'expressions, chaque expression du programme possédant une valeur décidée par la sémantique du langage.

- $\mathbb{Z}$ est l'ensemble des constantes entières.
- $\mathcal{V}ars$ est l'ensemble des variables considérées dans l'analyse statique.
- $\{*p \mid p \in \mathcal{V}ars^{ptr}\}$ est l'ensemble des déréférencements de pointeurs.
- $\{\&v \mid v \in \mathcal{V}ars^{mem}\}$ est l'ensemble des opérations d'adressage de variables.
- ⍰ est l'ensemble des expressions pour lesquelles l'analyse statique ne peut pas fournir la valeur, c'est-à-dire toutes les expressions qui n'entrent pas dans une des catégories ci-dessus. De fait, toutes les expressions incluant un calcul ou une opération sont dans cette catégorie, de même que les variables qui ne sont pas prises en compte ($\overline{\mathcal{V}ars} \subseteq$ ⍰).

En Coq, l'ensemble des expressions est un type algébrique. Nous ne définissons pas le type des opérations de déréférencements et de prise d'addresse car nous les prenons en compte en distinguant les types de nœuds (voir plus bas).

```
Variable OtherExprs : Type.

Inductive Exprs : Type :=
| int (z:Z)
| var (v:Vars)
| other (o: OtherExprs).
```

Les variables sont exploitées par l'analyse pour décider si un chemin est possible ou non. Pour qu'un chemin soit possible, il faut que tous les branchements conditionnels le composant puissent être pris au cours de la même exécution. Cela n'est possible que s'il existe une valuation possible pour les variables (par exemple, on ne doit pas avoir dans le chemin à la fois un nœud forçant la valeur d'une variable à 0 et un arc demandant que la même variable soit supérieure à 1). Nous définissons la notion de *contraintes* sur les variables pour exprimer les conditions portant sur la valeur des variables.

Définition 4 (Contraintes)**.** Les contraintes forment l'ensemble $\mathcal{C}$, elles décrivent des prédicats sur les valeurs des variables. Une contrainte est soit une relation entre deux

variables, soit une relation entre une variable entière et une constante, soit la contrainte spéciale `true` dénotant en fait l'absence de contrainte.

$$\begin{aligned}\mathcal{C} = &\{\texttt{true}\}\\ &\cup \mathcal{V}ars \times \{<, >, \leq, \geq, =, \neq\} \times \mathcal{V}ars\\ &\cup \mathcal{V}ars^{\mathbb{Z}} \times \{<, >, \leq, \geq, =, \neq\} \times \mathbb{Z}\end{aligned}$$

La définition Coq est similaire.

```
Inductive Ops : Type :=
| Eq
| Neq
| Leq
| Lt
| Geq
| Gt.

Inductive Constraint : Type :=
| constraint1 (v:Vars) (o:Ops) (z:Z)
| constraint2 (v:Vars) (o:Ops) (v':Vars)
| true.

Lemma Constraint_eq_dec :∀c c':Constraint, {c = c'}+{c ≠ c'}.
```

Définition 5 (Nœuds)**.** Un nœud est un élement de l'ensemble

$$\mathcal{V} = \mathcal{V}^{assign} \uplus \mathcal{V}^{mem} \uplus \mathcal{V}^{join} \uplus \mathcal{V}^{\varphi} \uplus \mathcal{V}^{call}$$

Les nœuds se présentent sous la syntaxe décrite dans le tableau suivant, où chaque énumération décrit une syntaxe possible. On note que nous traitons les adresses mémoire comme des constantes entières.

$\underline{\mathcal{V}^{assign}}$

$x = e$ avec :

- $x \in \mathcal{V}ars^{\mathbb{Z}}$ et $e \in \mathbb{Z} \cup \mathcal{V}ars^{\mathbb{Z}} \cup \mathcal{V}ars^{ptr} \cup \{*p \mid p \in \mathcal{V}ars^{ptr}\} \cup \{?\}$
- $x \in \mathcal{V}ars^{ptr}$ et $e \in \mathbb{Z} \cup \mathcal{V}ars^{\mathbb{Z}} \cup \mathcal{V}ars^{ptr} \cup \{*p \mid p \in \mathcal{V}ars^{ptr}\} \cup \{\&y \mid y \in \mathcal{V}ars^{mem}\} \cup \{?\}$
- $w \in \overline{\mathcal{V}ars}$ et $e \in \mathbb{Z} \cup \mathcal{V}ars^{\mathbb{Z}} \cup \mathcal{V}ars^{ptr} \cup \{*p \mid p \in \mathcal{V}ars^{ptr}\} \cup \{\&y \mid y \in \mathcal{V}ars^{mem}\} \cup \{?\}$

$\underline{\mathcal{V}^{mem}}$

$*p = e$ avec :

- $p \in \mathcal{V}ars^{ptr}$ et $e \in \mathbb{Z} \cup \mathcal{V}ars^{\mathbb{Z}} \cup \mathcal{V}ars^{ptr} \cup \{*p \mid p \in \mathcal{V}ars^{ptr}\} \cup \{\&y \mid y \in \mathcal{V}ars^{mem}\} \cup \{?\}$
- $p \in \overline{\mathcal{V}ars}$ et $e \in \mathbb{Z} \cup \mathcal{V}ars^{\mathbb{Z}} \cup \mathcal{V}ars^{ptr} \cup \{*p \mid p \in \mathcal{V}ars^{ptr}\} \cup \{\&y \mid y \in \mathcal{V}ars^{mem}\} \cup \{?\}$

$\underline{\mathcal{V}^{join}}$

Les nœuds de jonction n'ont pas de syntaxe particulière parce qu'ils ne correspondent pas à strictement parler à des instructions. Ils servent à simplifier la syntaxe du graphe. Étant donné un nœud de jonction v, on définit l'arité de v comme le nombre d'arcs entrants de v.

$\underline{\mathcal{V}^{\varphi}}$

$x = \texttt{PHI}\langle e_1, \ldots, e_n \rangle$ avec :

- $x \in \mathcal{V}ars^{\mathbb{Z}} \cap \mathcal{V}ars^{temp}$ et $\forall i \in \{1, \ldots, n\}$ $e_i \in \mathbb{Z} \cup \mathcal{V}ars^{\mathbb{Z}} \cup \{?\}$
- $x \in \mathcal{V}ars^{ptr} \cap \mathcal{V}ars^{temp}$ et $\forall i \in \{1, \ldots, n\}$ $e_i \in \mathbb{Z} \cup \mathcal{V}ars^{ptr} \cup \{\&y \mid y \in \mathcal{V}ars^{mem}\} \cup \{?\}$

On peut noter deux particularités importantes des nœuds ϕ par rapport aux autres nœuds d'affectation : la variable en partie gauche est toujours une variable temporaire et les expressions en partie droite ne sont jamais des opérations arithmétiques. En effet, GCC dans le cas des nœuds ϕ génère toujours les variables intermédiaires nécessaires pour respecter ces contraintes.

L'arité d'un nœud ϕ est définie comme le nombre d'expressions en partie droite du nœud $(e_1, \ldots, e_n)$. La propriété suivante est garantie par GCC : un nœud ϕ d'arité n dans le graphe est toujours précédé soit d'un autre nœud ϕ d'arité n, soit d'un nœud de choix d'arité n.

$\underline{\mathcal{V}^{call}}$

Les appels de fonction peuvent ou non retourner une valeur, qui peut être stockée dans une variable. Une fonction peut avoir ou non des arguments. Elle peut donc s'écrire :

$x = f()$ avec :

- $x \in \mathcal{V}ars$
- $x \in \overline{\mathcal{V}ars}$

ou

$f()$

ou

$x = f(e_1, \ldots, e_n)$ avec :

- $x \in \mathcal{V}ars$ et $\forall i \in \{1, \ldots, n\}$ $e_i \in \mathcal{V}ars \cup \mathbb{Z} \cup \{*p \mid p \in \mathcal{V}ars^{ptr}\} \cup \{\&y \mid y \in \mathcal{V}ars^{mem}\} \cup \{?\}$
- $x \in \overline{\mathcal{V}ars}$ et $\forall i \in \{1, \ldots, n\}$ $e_i \in \mathcal{V}ars \cup \mathbb{Z} \cup \{*p \mid p \in \mathcal{V}ars^{ptr}\} \cup \{\&y \mid y \in \mathcal{V}ars^{mem}\} \cup \{?\}$

ou

$f(e_1, \ldots, e_n)$ avec :

- $\forall i \in \{1, \ldots, n\}$ $e_i \in \mathcal{V}ars \cup \mathbb{Z} \cup \{*p \mid p \in \mathcal{V}ars^{ptr}\} \cup \{\&y \mid y \in \mathcal{V}ars^{mem}\} \cup \{?\}$

Nous définissons similairement en Coq tous les nœuds possibles à l'aide d'un type algébrique. Nous rétablissons ici les opérations de déréférencements et de prise d'adresse. Par exemple `assignPtr_to_Ptr` et `assignDeref_to_Ptr` semblent avoir le même type mais le premier cas correspond à une affectation de pointeur à pointeur (de type $p = q$) tandis que le second correspond à l'affectation du déréférencement d'un pointeur dans un autre (de type $p = *q$).

```
1 Inductive Nodes : Type :=
```

```
| assignZ_to_Varz (x:Vars) (Hnptr_x: ¬ pointer x) (e:Z)
| assignVarz_to_Varz (x:Vars) (Hnptr_x: ¬ pointer x) (e:Vars) (Hnptr_e: ¬ pointer e)
| assignPtr_to_Varz (x:Vars) (Hnptr_x: ¬ pointer x) (p:Vars) (Hptr_p: pointer p)
| assignOther_to_Varz (x:Vars) (Hnptr_x: ¬ pointer x) (e:OtherExprs)
| assignDeref_to_Varz (x:Vars) (Hnptr_x: ¬ pointer x) (e:Vars)
  (Hptr_e: pointer e) (He: pointers_to_non_pointers e Hptr_e)
| assignZ_to_Ptr (p:Vars) (Hptr_p: pointer p) (e:Z)
| assignVarz_to_Ptr (p:Vars) (Hptr_p: pointer p) (x:Vars) (Hnptr_x: ¬ pointer x)
| assignPtr_to_Ptr (p:Vars) (Hptr_p: pointer p) (q:Vars) (Hptr_q: pointer q)
  (Hconsistency:∀(v:Vars) (Haddr_v: addressable v),
    ptoracle q Hptr_q v Haddr_v → ptoracle p Hptr_p v Haddr_v)
| assignDeref_to_Ptr (p:Vars) (Hptr_p: pointer p) (q:Vars) (Hptr_q: pointer q)
  (Hq: pointers_to_pointers q Hptr_q)
  (Hconsistency:∀v Haddr_v Hptr_v, ptoracle q Hptr_q v Haddr_v →
    (forall x Haddr_x, ptoracle v Hptr_v x Haddr_x →
      ptoracle p Hptr_p x Haddr_x))
| assignAddr_to_Ptr (p:Vars) (Hptr_p: pointer p) (a:Vars)
  (Haddr_a: addressable a) (Hconsistency: ptoracle p Hptr_p a Haddr_a)
| assignOther_to_Ptr (p:Vars) (Hptr_p: pointer p) (e:OtherExprs)
| assign_to_IgnoredVar (w:IgnoredVars) (e:Exprs)
| memZ_to_Ptr (p:Vars) (Hptr_p: pointer p) (e:Z)
  (Hp: pointers_to_non_pointers p Hptr_p)
| memVarz_to_Ptr (p:Vars) (Hptr_p: pointer p) (x:Vars) (Hnptr_x: ¬ pointer x)
  (Hp: pointers_to_non_pointers p Hptr_p)
| memPtr_to_Ptr (p:Vars) (Hptr_p: pointer p) (q:Vars) (Hptr_q: pointer q)
  (Hp: pointers_to_pointers p Hptr_p)
  (Hconsistency:∀v Haddr_v Hptr_v, ptoracle p Hptr_p v Haddr_v →
    (forall x Haddr_x, ptoracle q Hptr_q x Haddr_x →
      ptoracle v Hptr_v x Haddr_x))
| memOther_to_Ptr (p:Vars) (Hptr_p: pointer p) (e:OtherExprs)
| mem_to_IgnoredVar (w:IgnoredVars) (e:Exprs)
| phiZ_to_Varz (x:Vars) (Hnptr_x: ¬ pointer x) (Hnaddr_x: ¬ addressable x) (e:Z)
| phiVarz_to_Varz (x:Vars) (Hnptr_x: ¬ pointer x) (Hnaddr_x: ¬ addressable x)
  (e:Vars) (Hnptr_e: ¬ pointer e)
| phiOther_to_Varz (x:Vars) (Hnptr_x: ¬ pointer x) (Hnaddr_x: ¬ addressable x)
  (e:OtherExprs)
| phiPtr_to_Ptr (p:Vars) (Hptr_p: pointer p) (Hnaddr_p: ¬ addressable p)
  (e:Vars) (Hptr_e: pointer e)
  (Hconsistency:∀v Haddr_v, ptoracle e Hptr_e v Haddr_v →
   ptoracle p Hptr_p v Haddr_v)
| phiAddr_to_Ptr (p:Vars) (Hptr_p: pointer p) (Hnaddr_p: ¬ addressable p)
  (y:Vars) (Haddr_y: addressable y)
  (Hconsistency: ptoracle p Hptr_p y Haddr_y)
| phiOther_to_Ptr (p:Vars) (Hptr_p: pointer p) (Hnaddr_p: ¬ addressable p)
  (e:OtherExprs)
| callVars (x:Vars)
| callOther (x:IgnoredVars)
| call
| join.
```

Définition 6 (Arcs)**.** L'ensemble des arcs est défini comme $\mathcal{E} = \mathcal{V} \times \mathcal{C} \times \mathcal{V}$.

Un arc est annoté par une contrainte dans $\mathcal{C}$ donnant la condition de franchissement. Les arcs sortants d'un même nœud portent des contraintes complémentaires.

En Coq, nous ne définissons pas le type des arcs car nous manipulons uniquement celui des contraintes.

5.4.2 Configurations et exécutions abstraites et concrètes

Il est impossible dans le cas général de décider statiquement si un chemin d'exécution est possible ou impossible car c'est une propriété non triviale du programme. C'est une conséquence du théorème de Rice [74, Corollaire B]. Cependant, il est possible de calculer une sur-approximation des chemins possibles et d'exclure de manière certaine beaucoup de chemins impossibles. Nous considérons deux sémantiques : la sémantique concrète du code de l'appel système et la sémantique abstraite qui s'applique aux graphes. Ces deux sémantiques sont écrites comme des relations de transitions entre états du système. Dans le cas de la sémantique abstraite, cet état est une configuration abstraite contenant des contraintes sur les variables. Dans le cas de la sémantique concrète, il s'agit de la valuation des variables. Nous modélisons les exécutions concrètes du noyau par des *exécutions abstraites* qui sont l'objet de notre analyse statique. Une exécution abstraite est une séquence de transitions permises par la sémantique abstraite et une *exécution concrète* est son pendant pour la sémantique concrète. Une exécution abstraite représente le parcours d'un chemin du graphe tandis que l'exécution concrète correspondante est l'enchaînement des instructions correspondant dans le noyau en activité. Naturellement, nous définissons la sémantique des exécutions abstraites en fonction de ce que nous savons de la véritable sémantique des appels système, de sorte qu'un chemin déclaré impossible par l'analyse statique corresponde à une exécution impossible du noyau. Il s'agit de la proposition 1 page 115.

Notre analyse consiste à étudier chaque chemin d'exécution abstraite afin de lui attribuer une configuration abstraite contenant des contraintes sur les variables. Pour prouver que notre analyse est correcte, la configuration abstraite d'une certaine exécution abstraite doit être compatible avec toute configuration concrète atteignable depuis une exécution concrète correspondante. Si la configuration abstraite est impossible (par exemple, elle contient des contraintes incompatibles entre elles) alors le chemin lui-même est impossible : il ne correspond à aucune exécution concrète possible.

Les configurations sont définies formellement comme suit.

Définition 7 (Configurations abstraite et concrète)**.** Les configurations représentent l'état du système lors d'une exécution, abstraite ou concrète.

Configuration abstraite Une configuration abstraite est un élément de l'ensemble $\mathcal{K} = \wp(\mathcal{C}) \times (\mathcal{V}ars^{ptr} \to \mathcal{V}ars^{mem} \cup \{\top\})$. C'est donc un couple (C, P) comprenant

- C un ensemble de contraintes de $\mathcal{C}$ sur les variables ;
- P une fonction d'alias, donnant pour chaque pointeur la variable sur laquelle il pointe si celle-ci peut être identifiée de manière univoque, ou la valeur spéciale $\top$ indiquant que la destination précise du pointeur n'est pas connue.

Configuration concrète Nous considérons que la mémoire du programme est un ensemble $Addrs$ de cases mémoire, contenant chacune exactement une valeur. Dans la réalité, les éléments de $Addrs$ sont un sous-ensemble des adresses de la mémoire, donc des entiers. Une configuration concrète est un triplet $(\gamma, \sigma, \alpha) \in \Theta$ où :

- $\gamma : \mathcal{V}ars^{temp} \to \mathbb{Z}$ est une fonction donnant pour chaque variable temporaire (les variables de $\mathcal{V}ars^{temp}$) sa valeur ;

- $\sigma : Addrs \to \mathbb{Z}$ est une fonction donnant pour chaque case mémoire la valeur de la variable qui y est stockée ;
- $\alpha : \mathcal{V}ars^{mem} \to Addrs$ est une fonction donnant pour chaque variable en mémoire (les variables de $\mathcal{V}ars^{mem}$) la case mémoire où elle est stockée. α est injective, seule une variable peut occuper un emplacement mémoire donné.

Dans la configuration abstraite, la fonction P est en fait un raffinement de l'oracle d'alias construit par le compilateur GCC pour ses propres besoins. Lorsque pour un pointeur p, on a $P(p) \neq \top$, cela signifie que notre analyse a un résultat plus précis que cet oracle, et connaît la destination exacte du pointeur p, parce qu'il tient compte du chemin d'exécution suivi.

Dans la configuration concrète, la première fonction γ donne la valeur des variables hors-mémoire. La seconde donne l'état mémoire du programme. Il est nécessaire de distinguer ces deux fonctions car en réalité, les variables de $\mathcal{V}ars^{temp}$ ne sont pas modifiées via des accès à la mémoire. En fait, bien qu'elles fassent partie du graphe de flot de contrôle, beaucoup d'entre elles sont optimisées par le compilateur avant de produire l'exécutable final. Dans tous les cas, elles ne sont jamais modifiées par un effet de bord ou via un pointeur. La troisième fonction enfin, donne l'organisation de la mémoire.

En Coq, nous définissons les configurations de manière similaire, à ceci près que la fonction d'alias P embarque la preuve de sa cohérence avec l'oracle d'alias $\mathcal{O}$, pour faciliter les démonstrations. Dans les configurations concrètes, nous divisons les fonctions γ et σ en deux selon que leur argument est un pointeur ou non, afin de faciliter les preuves et de simplifier le typage. Nous utilisons également l'hypothèse que la fonction α ne change jamais (les variables ne changent pas d'adresses une fois déclarées, ou du moins cela est transparent du point de vue du code) pour en faire une fonction à part de la configuration concrète.

```
Definition ConstraintSet : Type := set Constraint.
Inductive Target : Type :=
| ptvar (v:Vars) (Haddr: addressable v)
| ptunknown.
Definition PointerMap : Type :=∀(x:Vars) (Hptr: pointer x), Target.
Definition PointerMap_is_consistent P : Prop :=
forall (x:Vars) (Hptr: pointer x) (v:Vars) (Haddr: addressable v),
  P x Hptr = ptvar v Haddr → ptoracle x Hptr v Haddr.
Definition AbstractConfiguration : Type :=
 (ConstraintSet * (sig PointerMap_is_consistent))%type.

Variable MemoryLocation : Type.
Hypothesis MemoryLocation_eq_dec :∀l l':MemoryLocation, {l = l'}+{l ≠ l'}.
Definition GammaZ : Type :=∀(v:Vars) (Haddr_v: ¬addressable v), Z.
Definition GammaLoc : Type :=∀(v:Vars) (Haddr_v: ¬addressable v), MemoryLocation.
Definition SigmaZ : Type :=∀(l:MemoryLocation), Z.
Definition SigmaLoc : Type :=∀(l:MemoryLocation), MemoryLocation.
Variable alpha :∀(v:Vars) (Haddr_v: addressable v), MemoryLocation.
Hypothesis location_is_unique :∀x Haddr_x y Haddr_y, alpha x Haddr_x = alpha y
    Haddr_y → x = y.
Definition MemoryState : Type := (GammaZ * SigmaZ * GammaLoc * SigmaLoc)%type.
Definition gz (m:MemoryState) : GammaZ := fst (fst (fst m)).
Definition sz (m:MemoryState) : SigmaZ := snd (fst (fst m)).
```

```
Definition gloc (m:MemoryState) : GammaLoc := snd (fst m).
Definition sloc (m:MemoryState) : SigmaLoc := snd m.
```

Nous définissons également une fonction d'évaluation des expressions du code.

Définition 8 (Valuation d'une variable)**.** Étant donnée une configuration concrète $\theta = (\gamma, \sigma, \alpha) \in \Theta$ et une variable $x \in \mathcal{V}ars$, on définit $\theta(x)$ comme la valuation de x. On a :

$$\forall x \in \mathcal{V}ars\ \theta(x) = \begin{cases} \gamma(x) & x \in \mathcal{V}ars^{temp} \\ \sigma(\alpha(x)) & x \in \mathcal{V}ars^{mem} \end{cases}$$

$$\forall x \in \mathcal{V}ars^{ptr}\ \theta(x) \in \bigcup_{z \in \mathcal{V}ars^{mem}} \{\alpha(z)\} \cup \{0\}$$

avec $0 \in \mathcal{A}ddrs$ dénotant une valeur spéciale, celle du pointeur nul. Un pointeur dont la valeur est 0 ne pointe sur aucune variable. Dans un code C, un pointeur peut également pointer sur une adresse invalide (ne correspondant à aucune variable). Nous faisons abstraction de ce cas (qui correspond en réalité à une erreur de programmation) et considérons que tous les pointeurs qui ne sont pas déréférençables correspondent à la valeur 0.

Nous définissons de manière naturelle la fonction de valuation en Coq et nous formalisons l'hypothèse selon laquelle un pointeur ne peut pointer sur une certaine variable que si l'oracle d'alias le permet.

```
Definition Valuation (theta:MemoryState) (v:Vars) : Z + MemoryLocation.
destruct (Addressable_dec v).
- destruct (Pointer_dec v).
  + exact (inr (sloc theta (alpha v a))).
  + exact (inl (sz theta (alpha v a))).
- destruct (Pointer_dec v).
  + exact (inr (gloc theta v n)).
  + exact (inl (gz theta v n)).
Defined.

Hypothesis Valuation_is_consistent :
 ∀(theta:MemoryState) (p:Vars) (Hptr_p: pointer p)
   (y:Vars) (Haddr_y: addressable y),
  Valuation theta p = inr (alpha y Haddr_y) → ptoracle p Hptr_p y Haddr_y.
```

Définition 9 (Évaluation d'une expression)**.** Nous supposons l'existence d'une fonction $[\![\cdot]\!] : \mathcal{V}ars \cup \mathbb{Z} \cup \{*p \mid p \in \mathcal{V}ars^{ptr}\} \cup \{\&x \mid x \in \mathcal{V}ars^{mem}\} \cup \textcircled{?} \to \mathbb{Z} \cup \mathcal{A}ddrs$ associant à chaque expression une valeur dans la sémantique concrète du langage telle que :

$$\begin{aligned} [\![k]\!]^\theta &= k \in \mathbb{Z} && \text{où } k \text{ est un entier} \\ [\![x]\!]^\theta &= \theta(x) && \text{où } x \in \mathcal{V}ars \\ [\![*p]\!]^\theta &= \sigma([\![p]\!]^\theta) && \text{où } p \in \mathcal{V}ars^{ptr} \\ [\![\&x]\!]^\theta &= \alpha([\![x]\!]^\theta) && \text{où } x \in \mathcal{V}ars^{mem} \end{aligned}$$

On note que nous ne faisons pas d'hypothèse sur la valuation des expressions de ⍰ ; en revanche, on suppose que leur évaluation termine et ne provoque pas d'écriture dans la mémoire.

Comme nous n'avons pas défini à part les opérations de déréférencement et de prise d'adresse, nous n'avons pas besoin d'une fonction supplémentaire de valuation des expressions : la valuation d'un entier est cet entier lui-même, la valuation d'une variable est donnée par la fonction présentée plus haut et la valuation d'une expression de ⍰ est inconnue.

Pour exprimer le fait qu'une configuration abstraite est un modèle correct d'une configuration concrète, nous définissons tout d'abord une relation de compatibilité entre une configuration concrète et une contrainte.

Définition 10 (Compatibilité entre configurations)**.** Une configuration concrète $\theta \in \Theta$ est compatible avec une contrainte $c \in \mathcal{C}$, ce que l'on note $\theta \vDash c$ si, et seulement si :

$$
\begin{aligned}
& c = \texttt{true} \\
\vee \quad & c = (x, \lessgtr, y) \in \mathcal{V}ars^{\mathbb{Z}} \times \{<, >, \leq, \geq, =, \neq\} \times \mathcal{V}ars^{\mathbb{Z}} \wedge \theta(x) \lessgtr \theta(y) \\
\vee \quad & c = (x, \lessgtr, n) \in \mathcal{V}ars^{\mathbb{Z}} \times \{<, >, \leq, \geq, =, \neq\} \times \mathbb{Z} \wedge \theta(x) \lessgtr n \\
\vee \quad & c = (x, \lessgtr, y) \in \mathcal{V}ars^{ptr} \times \{<, >, \leq, \geq, =, \neq\} \times \mathcal{V}ars^{ptr} \wedge \theta(x) \lessgtr \theta(y)
\end{aligned}
$$

avec $\lessgtr$ l'interprétation usuelle de l'opérateur pris dans $\{<, >, \leq, \geq, =, \neq\}$.

Nous étendons cette relation aux ensembles de contraintes, en abusant légèrement des notations.

$$\theta \vDash C \Leftrightarrow \bigwedge_{c \in C} \theta \vDash c$$

En Coq, nous définissons une propriété nommée `satisfiability` pour exprimer la compatibilité entre une configuration concrète et une contrainte ou un ensemble de contrainte.

```
Inductive satisfiability : MemoryState → Constraint → Prop :=
| satisfy_true theta:
  satisfiability theta true
| satisfy_constraint_Eq_Z theta x k (Hval: Valuation theta x = inl k):
  satisfiability theta (constraint1 x Eq k)
| satisfy_constraint_Neq_Z theta x k1 k2 (Hval: Valuation theta x = inl k2)
  (Hneq: k1 ≠ k2):
  satisfiability theta (constraint1 x Neq k2)
| satisfy_constraint_Leq_Z theta x k1 k2 (Hval: Valuation theta x = inl k2)
  (Hneq: k1 <= k2):
  satisfiability theta (constraint1 x Leq k2)
| satisfy_constraint_Lt_Z theta x k1 k2 (Hval: Valuation theta x = inl k2)
  (Hneq: k1 < k2):
  satisfiability theta (constraint1 x Lt k2)
| satisfy_constraint_Geq_Z theta x k1 k2 (Hval: Valuation theta x = inl k2)
  (Hneq: k1 >= k2):
  satisfiability theta (constraint1 x Geq k2)
| satisfy_constraint_Gt_Z theta x k1 k2 (Hval: Valuation theta x = inl k2)
  (Hneq: k1 > k2):
  satisfiability theta (constraint1 x Gt k2)
```

```
| satisfy_constraint_Eq_Var theta x y
  (Hval: Valuation theta x = Valuation theta y):
  satisfiability theta (constraint2 x Eq y)
| satisfy_constraint_Neq_Var theta x y k1 k2
  (Hvalx: Valuation theta x = inl k1) (Hvaly: Valuation theta y = inl k2)
  (Hneq: k1 ≠ k2):
  satisfiability theta (constraint2 x Neq y)
| satisfy_constraint_Leq_Var theta x y k1 k2
  (Hvalx: Valuation theta x = inl k1) (Hvaly: Valuation theta y = inl k2)
  (Hneq: k1 <= k2):
  satisfiability theta (constraint2 x Leq y)
| satisfy_constraint_Lt_Var theta x y k1 k2
  (Hvalx: Valuation theta x = inl k1) (Hvaly: Valuation theta y = inl k2)
  (Hneq: k1 < k2):
  satisfiability theta (constraint2 x Lt y)
| satisfy_constraint_Geq_Var theta x y k1 k2
  (Hvalx: Valuation theta x = inl k1) (Hvaly: Valuation theta y = inl k2)
  (Hneq: k1 >= k2):
  satisfiability theta (constraint2 x Geq y)
| satisfy_constraint_Gt_Var theta x y k1 k2
  (Hvalx: Valuation theta x = inl k1) (Hvaly: Valuation theta y = inl k2)
  (Hneq: k1 > k2):
  satisfiability theta (constraint2 x Gt y).

Definition satisfiability_set_constraints theta C :=
∀c, set_In c C → satisfiability theta c.
```

5.4.3 Sémantiques abstraite et concrète

Nous supposons qu'il existe une sémantique concrète du code, fournie par le compilateur, pouvant s'écrire sous la forme d'une relation de transition entre configurations concrètes.

$$\rightarrow \subseteq \Theta \times \mathcal{V} \times \mathcal{C} \times \Theta.$$

Intuitivement, si $(\theta_1, v, c, \theta_2) \in \rightarrow$, ce que l'on note $\theta_1 \xrightarrow{v,c} \theta_2$, cela signifie que dans l'exécution concrète considérée, on passe de la configuration θ_1 à la configuration θ_2 en passant du nœud v à son successeur par le franchissement de l'arc labellisé par c, comme illustré dans la figure 5.3.

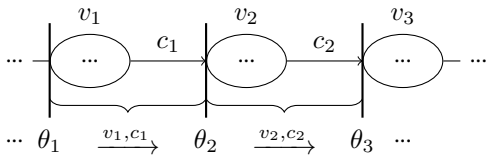


FIGURE 5.3 – Fonctionnement de la sémantique concrète : une transition comprend un nœud plus un arc

Nous ne donnons pas une définition complète de cette sémantique mais nous faisons seulement certaines hypothèses simples et communes sur cette sémantique, listées ci-dessous.

Nous exprimons les propriétés portant sur la relation $\rightarrow$ en distinguant chaque type de nœud.

- $v \in \mathcal{V}^{assign}$ **de la forme** $x = e$**, avec** $x \in \mathcal{V}ars^{temp}$

$$\theta = (\gamma, \sigma, \alpha) \xrightarrow{v,c} (\gamma[x \leftarrow [\![e]\!]^{\theta}], \sigma, \alpha)$$

  Une affectation simple assigne à la variable x la valeur de l'expression e. Avec $x \in \mathcal{V}ars^{temp}$, c'est la fonction γ de la configuration concrète qui est modifiée.

- $v \in \mathcal{V}^{assign}$ **de la forme** $x = e$**, avec** $x \in \mathcal{V}ars^{mem}$

$$\theta = (\gamma, \sigma, \alpha) \xrightarrow{v,c} (\gamma, \sigma[\alpha(x) \leftarrow [\![e]\!]^{\theta}], \alpha)$$

  Avec $x \in \mathcal{V}ars^{mem}$, c'est la fonction σ qui est modifiée. $\alpha(x)$ ici donne l'adresse de x.

- $v \in \mathcal{V}^{mem}$ **de la forme** $*p = e$

$$\theta = (\gamma, \sigma, \alpha) \xrightarrow{v,c} (\gamma, \sigma[[\![p]\!]^{\theta} \leftarrow [\![e]\!]^{\theta}], \alpha)$$

  Une affection via un pointeur met la valeur de l'expression e dans la variable pointée par p, qui est nécessairement une variable en mémoire dans $\mathcal{V}ars^{mem}$. C'est donc la fonction σ qui est modifiée.

- $v \in \mathcal{V}^{\varphi}$ **de la forme** $x = \texttt{PHI}\langle e_1, \ldots, e_n \rangle$

$$\theta = (\gamma, \sigma, \alpha) \xrightarrow{v,c} (\gamma[x \leftarrow [\![e_{n_{path}}]\!]^{\theta}], \sigma, \alpha)$$

  où n_{path} est l'indice de l'arc pris pour atteindre le dernier nœud de jonction (cet indice est maintenu le long du parcours du chemin, en numérotant les arcs de chaque nœud de jonction).

  Une affectation dans un nœud ϕ met la valeur d'une des expressions en partie droite, choisie en fonction du chemin suivi, dans x, qui est toujours une valeur de $\mathcal{V}ars^{temp}$ (cette restriction est appliquée par le compilateur). Il est toujours possible de connaître n_{path} dans notre cas puisque nous analysons les chemins un par un.

- $v \in \mathcal{V}^{call}$ **de la forme** $x = f(e_1, \ldots, e_n)$

$$(\gamma, \sigma, \alpha) \xrightarrow{v,c} (\gamma_2, \sigma_2, \alpha_2)$$

  où $\forall a \in \mathcal{V}ars^{temp} \setminus \{x\}\ \gamma_2(a) = \gamma(a)$ et σ_2 et α_2 représentent un état mémoire sur lequel aucune hypothèse n'est faite.

  Un appel de fonction (ou l'exécution d'une portion de code en assembleur) est susceptible de changer les valeurs de toutes les variables de $\mathcal{V}ars^{mem}$ car elles pourraient être affectées, y compris via un pointeur, dans la fonction appelée. En revanche, les variables de $\mathcal{V}ars^{temp}$ sont protégées car le compilateur les maintient strictement séparées pour pouvoir appliquer ses propres optimisations. On obtient donc comme configuration concrète

résultante un état mémoire arbitrairement différent et une fonction γ identique à celle de départ, sauf en ce qui concerne la variable résultat (toujours une variable de $\mathcal{V}ars^{temp}$) qui peut contenir une valeur arbitraire au retour de la fonction.

— $v \in \mathcal{V}^{call}$ **de la forme** $f(e_1, \ldots, e_n)$

$$(\gamma, \sigma, \alpha) \xrightarrow{v,c} (\gamma, \sigma_2, \alpha_2)$$

où σ_2 et α_2 représentent un état mémoire sur lequel aucune hypothèse n'est faite.

La situation est similaire dans le cas où il n'y a pas de variable de retour.

— $v \in \mathcal{V}^{join}$

$$\theta \xrightarrow{v,c} \theta$$

Les nœuds de choix ne changent pas la configuration courante car ils ne correspondent à aucune instruction matérielle du programme terminée, ils appartiennent seulement à la syntaxe du graphe.

On a également une propriété sur les arcs. Une exécution ne peut pas emprunter un arc si la condition de franchissement n'est pas vérifiée dans la configuration concrète.

$$\forall \theta_1, \theta_2 \in \Theta \; \forall v \in \mathcal{V} \; \forall c \in \mathcal{C} \; \theta_1 \xrightarrow{v,c} \theta_2 \Rightarrow \theta_2 \vDash c \tag{5.1}$$

En Coq, nous suivons le même raisonnement. Nous supposons en premier lieu l'existence d'une fonction transformant une configuration concrète en une autre au passage d'un nœud puis nous définissons une relation entre configurations concrètes conditionnée par un nœud et une contrainte (portée par l'arc suivant le nœud). La relation est vérifiée si la configuration finale est le résultat de la mise à jour de la configuration initiale au passage du nœud et qu'elle est compatible avec la contrainte. Cela modélise le fait qu'une expression ne peut emprunter un branchement conditionnel que si la condition est respectée. Nous listons ensuite les hypothèses qui s'appliquent à chaque type de nœud.

```
Variable ConcreteSemantics' : MemoryState → Nodes → MemoryState.
Inductive ConcreteSemantics : MemoryState → Nodes → Constraint →
    MemoryState → Prop :=
 ConcreteSemanticsDef theta v c (Hsatis_c: satisfiability (ConcreteSemantics'
    theta v) c):
 ConcreteSemantics theta v c (ConcreteSemantics' theta v).

Hypothesis csem_assignZ_addr :
 ∀theta (x: Vars) (Hnptr_x: ¬pointer x) (Haddr_x: addressable x) z,
 ConcreteSemantics' theta (assignZ_to_Varz x Hnptr_x z) =
  update_sz theta (alpha x Haddr_x) z.

Hypothesis csem_assignZ_naddr :
 ∀theta (x: Vars) (Hnptr_x: ¬pointer x) (Hnaddr_x: ¬addressable x) z,
 ConcreteSemantics' theta (assignZ_to_Varz x Hnptr_x z) =
  update_gz theta x Hnptr_x Hnaddr_x z.
```

```
Hypothesis csem_assignVarz_addr :
 ∀theta (x: Vars) (Hnptr_x: ¬pointer x) (Haddr_x: addressable x)
   (e: Vars) (Hnptr_e: ¬pointer e),
  ConcreteSemantics' theta (assignVarz_to_Varz x Hnptr_x e Hnptr_e) =
   update_sz theta (alpha x Haddr_x) (ValuationZ theta e Hnptr_e).

Hypothesis csem_assignVarz_naddr :
 ∀theta (x: Vars) (Hnptr_x: ¬pointer x) (Hnaddr_x: ¬addressable x)
   (e: Vars) (Hnptr_e: ¬pointer e),
  ConcreteSemantics' theta (assignVarz_to_Varz x Hnptr_x e Hnptr_e) =
   update_gz theta x Hnptr_x Hnaddr_x (ValuationZ theta e Hnptr_e).

Hypothesis csem_assignPtr_to_Varz :
 ∀theta (x:Vars) (Hnptr_x: ¬pointer x) (e: Vars) (Hptr_e: pointer e),
 ∃theta2,
  ConcreteSemantics' theta (assignPtr_to_Varz x Hnptr_x e Hptr_e) = theta2 ∧
  (forall (y:Vars), x ≠ y → Valuation theta y = Valuation theta2 y).

Hypothesis csem_assignOther_to_Varz :
 ∀theta (x:Vars) (Hptr_x: ¬pointer x) (e: OtherExprs),
 ∃theta2,
  ConcreteSemantics' theta (assignOther_to_Varz x Hptr_x e) = theta2 ∧
  (forall (y:Vars), x ≠ y → Valuation theta y = Valuation theta2 y).

Hypothesis csem_assignDeref_to_Varz_addr :
 ∀theta (x:Vars) (Hnptr_x: ¬pointer x) (Haddr_x: addressable x)
   (e:Vars) (Hptr_e: pointer e)
   (He: pointers_to_non_pointers e Hptr_e),
  ConcreteSemantics' theta (assignDeref_to_Varz x Hnptr_x e Hptr_e He) =
   update_sz theta (alpha x Haddr_x) ((sz theta) (ValuationLoc theta e Hptr_e)).

Hypothesis csem_assignDeref_to_Varz_naddr :
 ∀theta (x:Vars) (Hnptr_x: ¬pointer x) (Hnaddr_x: ¬addressable x)
   (e:Vars) (Hptr_e: pointer e)
   (He: pointers_to_non_pointers e Hptr_e),
  ConcreteSemantics' theta (assignDeref_to_Varz x Hnptr_x e Hptr_e He) =
   update_gz theta x Hnptr_x Hnaddr_x ((sz theta) (ValuationLoc theta e Hptr_e)).

Hypothesis csem_assignZ_to_Ptr :
 ∀theta (x:Vars) (Hptr_x: pointer x) (z:Z),
 ∃theta2,
  ConcreteSemantics' theta (assignZ_to_Ptr x Hptr_x z) = theta2 ∧
   ∀(y:Vars), x ≠ y → Valuation theta y = Valuation theta2 y.

Hypothesis csem_assignVarz_to_Ptr :
 ∀theta (x:Vars) (Hptr_x: pointer x) (e:Vars) (Hnptr_e: ¬pointer e),
 ∃theta2,
  ConcreteSemantics' theta (assignVarz_to_Ptr x Hptr_x e Hnptr_e) = theta2 ∧
   ∀(y:Vars), x ≠ y → Valuation theta y = Valuation theta2 y.

Hypothesis csem_assignPtr_to_Ptr_addr :
 ∀theta (p:Vars) (Hptr_p: pointer p) (Haddr_p: addressable p)
   (q:Vars) (Hptr_q: pointer q) Hconsistency,
  ConcreteSemantics' theta (assignPtr_to_Ptr p Hptr_p q Hptr_q Hconsistency) =
```

```
  update_sloc theta (alpha p Haddr_p) (ValuationLoc theta q Hptr_q).

Hypothesis csem_assignPtr_to_Ptr_naddr :
∀theta (p:Vars) (Hptr_p: pointer p) (Hnaddr_p: ¬addressable p)
  (q:Vars) (Hptr_q: pointer q) Hconsistency,
 ConcreteSemantics' theta (assignPtr_to_Ptr p Hptr_p q Hptr_q Hconsistency) =
  update_gloc theta p Hptr_p Hnaddr_p (ValuationLoc theta q Hptr_q).

Hypothesis csem_assignDeref_to_Ptr_addr :
∀theta (p: Vars) (Hptr_p: pointer p) (Haddr_p: addressable p)
  (q:Vars) (Hptr_q: pointer q) Hpointers_q Hconsistency,
 ConcreteSemantics' theta (assignDeref_to_Ptr p Hptr_p q Hptr_q Hpointers_q
     Hconsistency) =
  update_sloc theta (alpha p Haddr_p) ((sloc theta) (ValuationLoc theta q Hptr_q)
     ).

Hypothesis csem_assignDeref_to_Ptr_naddr :
∀theta (p: Vars) (Hptr_p: pointer p) (Hnaddr_p: ¬addressable p)
  (q:Vars) (Hptr_q: pointer q) Hpointers_q Hconsistency,
 ConcreteSemantics' theta (assignDeref_to_Ptr p Hptr_p q Hptr_q Hpointers_q
     Hconsistency) =
  update_gloc theta p Hptr_p Hnaddr_p ((sloc theta) (ValuationLoc theta q Hptr_q
     )).

Hypothesis csem_assignAddr_to_Ptr_addr :
∀theta (p:Vars) (Hptr_p: pointer p) (Haddr_p: addressable p)
  (a:Vars) (Haddr_a: addressable a) Hconsistency,
 ConcreteSemantics' theta (assignAddr_to_Ptr p Hptr_p a Haddr_a Hconsistency)
     =
  update_sloc theta (alpha p Haddr_p) (alpha a Haddr_a).

Hypothesis csem_assignAddr_to_Ptr_naddr :
∀theta (p:Vars) (Hptr_p: pointer p) (Hnaddr_p: ¬addressable p)
  (a:Vars) (Haddr_a: addressable a) Hconsistency,
 ConcreteSemantics' theta (assignAddr_to_Ptr p Hptr_p a Haddr_a Hconsistency)
     =
  update_gloc theta p Hptr_p Hnaddr_p (alpha a Haddr_a).

Hypothesis csem_assignOther_to_Ptr :
∀theta (p:Vars) (Hptr_p: pointer p) (e: OtherExprs),
∃theta2,
 ConcreteSemantics' theta (assignOther_to_Ptr p Hptr_p e) = theta2 ∧
  ∀y, p ≠ y → Valuation theta y = Valuation theta2 y.

Hypothesis csem_assign_to_IgnoredVar :
∀theta (w:IgnoredVars) (e:Exprs),
 ConcreteSemantics' theta (assign_to_IgnoredVar w e) = theta.

Hypothesis csem_memZ_to_Ptr :
∀theta (p:Vars) (Hptr_p: pointer p) (e:Z) Hpointers_p,
 ConcreteSemantics' theta (memZ_to_Ptr p Hptr_p e Hpointers_p) =
  update_sz theta (ValuationLoc theta p Hptr_p) e.

Hypothesis csem_memVarz_to_Ptr :
```

```
∀theta (p:Vars) (Hptr_p: pointer p) (e:Vars) (Hnptr_e: ¬pointer e)
 Hpointers_p,
ConcreteSemantics' theta (memVarz_to_Ptr p Hptr_p e Hnptr_e Hpointers_p) =
 update_sz theta (ValuationLoc theta p Hptr_p) (ValuationZ theta e Hnptr_e).

Hypothesis csem_memPtr_to_Ptr :
∀theta (p:Vars) (Hptr_p: pointer p) (q:Vars) (Hptr_q: pointer q)
 Hpointers_p Hconsistency,
ConcreteSemantics' theta (memPtr_to_Ptr p Hptr_p q Hptr_q Hpointers_p
    Hconsistency) =
 update_sloc theta (ValuationLoc theta p Hptr_p) (ValuationLoc theta q Hptr_q).

Hypothesis csem_memOther_to_Ptr :
∀theta (p:Vars) (Hptr_p: pointer p) (e:OtherExprs),
∃theta2,
ConcreteSemantics' theta (memOther_to_Ptr p Hptr_p e) = theta2 ∧
 ∀(y:Vars),
  (¬ addressable y ∨ (forall (Haddr_y: addressable y),
   alpha y Haddr_y ≠ ValuationLoc theta p Hptr_p)) →
    Valuation theta y = Valuation theta2 y.

Hypothesis csem_mem_to_IgnoredVar :
∀theta (w:IgnoredVars) (e:Exprs),
ConcreteSemantics' theta (mem_to_IgnoredVar w e) = theta.

Hypothesis csem_phiZ_naddr :
∀theta (x: Vars) (Hnptr_x: ¬pointer x) (Hnaddr_x: ¬addressable x) z,
ConcreteSemantics' theta (phiZ_to_Varz x Hnptr_x Hnaddr_x z) =
 update_gz theta x Hnptr_x Hnaddr_x z.

Hypothesis csem_phiVarz_naddr :
∀theta (x: Vars) (Hnptr_x: ¬pointer x) (Hnaddr_x: ¬addressable x)
  (e: Vars) (Hnptr_e: ¬pointer e),
ConcreteSemantics' theta (phiVarz_to_Varz x Hnptr_x Hnaddr_x e Hnptr_e) =
 update_gz theta x Hnptr_x Hnaddr_x (ValuationZ theta e Hnptr_e).

Hypothesis csem_phiOther_to_Varz :
∀theta (x: Vars) (Hnptr_x: ¬pointer x) (Hnaddr_x: ¬addressable x)
  (e: OtherExprs),
∃theta2,
ConcreteSemantics' theta (phiOther_to_Varz x Hnptr_x Hnaddr_x e) = theta2 ∧
 ∀(y:Vars), x ≠ y → Valuation theta y = Valuation theta2 y.

Hypothesis csem_phiPtr_to_Ptr_naddr :
∀theta (p: Vars) (Hptr_p: pointer p) (Hnaddr_p: ¬addressable p)
  (q:Vars) (Hptr_q: pointer q) Hconsistency,
ConcreteSemantics' theta (phiPtr_to_Ptr p Hptr_p Hnaddr_p q Hptr_q
    Hconsistency) =
 update_gloc theta p Hptr_p Hnaddr_p (ValuationLoc theta q Hptr_q).

Hypothesis csem_phiAddr_to_Ptr_naddr :
∀theta (p: Vars) (Hptr_p: pointer p) (Hnaddr_p: ¬addressable p)
  (a:Vars) (Haddr_a: addressable a) Hconsistency,
```

```
 ConcreteSemantics' theta (phiAddr_to_Ptr p Hptr_p Hnaddr_p a Haddr_a
     Hconsistency) =
  update_gloc theta p Hptr_p Hnaddr_p (alpha a Haddr_a).

Hypothesis csem_phiOther_to_Ptr :
 ∀theta (p: Vars) (Hptr_p: pointer p) (Hnaddr_p: ¬addressable p)
   (e: OtherExprs),
 ∃theta2,
 ConcreteSemantics' theta (phiOther_to_Ptr p Hptr_p Hnaddr_p e) = theta2 ∧
  ∀y, p ≠ y → Valuation theta y = Valuation theta2 y.

Hypothesis csem_callVars :
 ∀theta (ret: Vars),
 ∃theta2,
 ConcreteSemantics' theta (callVars ret) = theta2 ∧
  ∀y (Hnaddr_y: ¬addressable y), y ≠ ret → Valuation theta y = Valuation theta2
     y.

Hypothesis csem_callOther :
 ∀theta (ret: IgnoredVars),
 ∃theta2,
 ConcreteSemantics' theta (callOther ret) = theta2 ∧
  ∀y (Hnaddr_y: ¬addressable y), Valuation theta y = Valuation theta2 y.

Hypothesis csem_call :
 ∀theta,
 ∃theta2,
 ConcreteSemantics' theta call = theta2 ∧
  ∀y (Hnaddr_y: ¬addressable y), Valuation theta y = Valuation theta2 y.

Hypothesis csem_join :
 ∀theta,
 ConcreteSemantics' theta join = theta.
```

Nous définissons en revanche complètement la sémantique abstraite des chemins du graphe. Cette sémantique est simple car elle vise uniquement à répondre aux objectifs de notre analyse qui est de distinguer les chemins impossibles parmi tous ceux qui esquivent les crochets LSM. Elle omet de prendre en compte beaucoup de traits du langage. La sémantique est une relation de transition entre configurations abstraites.

$$\twoheadrightarrow \subseteq \mathcal{K} \times \mathcal{V} \times \mathcal{C} \times \mathcal{K}$$

De même que pour la sémantique concrète, $k_1 \xrightarrow{v,c}\!\!\!\!\!\rightarrow\; k_2$ signifie que l'on passe de la configuration k_1 à la configuration k_2 en passant du nœud v à son successeur par l'arc étiqueté par la contrainte c.

Nous donnons la définition de $\twoheadrightarrow$ cas par cas, pour chaque type de nœud.

Nous définissons tout d'abord une fonction auxiliaire

$$\text{reset} : \wp(\mathcal{C}) \times \mathcal{V}ars \to \wp(\mathcal{C})$$

qui prend en paramètre un ensemble de contraintes et une variable et retourne cet

ensemble de contraintes privé de toutes celles portant sur la variable en paramètre, soit de manière intuitive :

$$\mathrm{reset}(C, x) = C \setminus \{(x, \cdot, \cdot), (\cdot, \cdot, x)\}$$

En abusant des notations, on étend cette fonction aux ensembles de variables :

$$\mathrm{reset}(C, X) = C \setminus (\{(x, \cdot, \cdot) \mid x \in X\} \cup \{(\cdot, \cdot, x) \mid x \in X\})$$

$\underline{\mathcal{V}^{assign}}$

- $v \in \mathcal{V}^{assign}$ **de la forme** $x = \alpha$ **avec** $x \in \mathcal{V}ars^{\mathbb{Z}}$**,** $\alpha \in \mathcal{V}ars^{\mathbb{Z}} \cup \mathbb{Z}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\mathrm{reset}(C, x) \cup \{(x, =, \alpha)\} \cup \{c\}, P)$$

  Lors d'une affectation simple d'une variable entière avec une autre variable ou une constante entière, on retire de la configuration abstraite toutes les contraintes portant sur x et on ajoute une nouvelle contrainte d'égalité. La fonction P donnant pour chaque pointeur sa destination n'est pas modifiée. On ajoute également la contrainte portée par l'arc qui suit le nœud, comme dans toutes les règles.

- $v \in \mathcal{V}^{assign}$ **de la forme** $p = q$ **avec** $p \in \mathcal{V}ars^{ptr}$**,** $q \in \mathcal{V}ars^{ptr}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\mathrm{reset}(C, p) \cup \{(p, =, q)\} \cup \{c\}, \\ P[p \leftarrow P(q)])$$

  Cette règle présente un cas similaire au précédent mais cette fois, un pointeur se voit affecté la valeur d'un autre pointeur. On applique les mêmes modifications de la configuration que dans la règle précédente et on modifie en plus la fonction P pour indiquer que p a à présent la même destination que q.

- $v \in \mathcal{V}^{assign}$ **de la forme** $p = \&v$ **avec** $p \in \mathcal{V}ars^{ptr}$**,** $v \in \mathcal{V}ars^{mem}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\mathrm{reset}(C, p) \cup \{c\}, P[p \leftarrow v])$$

  Dans le cas où un pointeur p se voit affecté l'adresse d'une variable v, on supprime les contraintes sur p mais on change la fonction P pour noter la nouvelle association entre p et v.

- $v \in \mathcal{V}^{assign}$ **de la forme** $x = *q$ **avec** $x \in \mathcal{V}ars^{\mathbb{Z}}$**,** $q \in \mathcal{V}ars^{ptr}$ **et** $P(q) = z \in \mathcal{V}ars^{\mathbb{Z}}$ **(et donc** $P(q) \neq \top$**)**

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\mathrm{reset}(C, x) \cup \{(x, =, z)\} \cup \{c\}, P)$$

  Si on assigne à une variable entière le déréférencement d'un pointeur q, et si la destination $P(q)$ de ce pointeur est connue et est un entier alors on peut ajouter la contrainte d'égalité entre la variable et la variable pointée.

- $v \in \mathcal{V}^{assign}$ **de la forme** $x = *q$ **avec** $x \in \mathcal{V}ars^{\mathbb{Z}}$**,** $q \in \mathcal{V}ars^{ptr}$ **et** $P(q) = z \in \mathcal{V}ars^{ptr}$ **ou** $P(q) = \top$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, x) \cup \{c\}, P)$$

En revanche, si la destination $P(q)$ du pointeur est un pointeur (transtypage de pointeur vers entier) ou bien est inconnue alors on retire seulement les contraintes sur x car sa nouvelle valeur est inconnue.

- $v \in \mathcal{V}^{assign}$ **de la forme** $p = *q$**,** $p \in \mathcal{V}ars^{ptr}$**,** $q \in \mathcal{V}ars^{ptr}$ **avec** $P(q) = r \in \mathcal{V}ars^{ptr}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, p) \cup \{(p, =, r)\} \cup \{c\}, P[p \leftarrow r])$$

Lorsqu'un pointeur est affecté le déréférencement d'un pointeur de pointeur, dont la valeur pointée est connue dans P, on met à jour l'ensemble de contraintes avec la nouvelle égalité ainsi que P.

- $v \in \mathcal{V}^{assign}$ **de la forme** $p = *q$ **avec** $p \in \mathcal{V}ars^{ptr}$**,** $q \in \mathcal{V}ars^{ptr}$ **et** $P(q) = \top$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, p) \cup \{c\}, P[p \leftarrow \top])$$

- $v \in \mathcal{V}^{assign}$ **de la forme** $x = e$ **avec** $x \in \mathcal{V}ars^{\mathbb{Z}}$ **et** $e = \textcircled{?} \cup \mathcal{V}ars^{ptr}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, x) \cup \{c\}, P)$$

Lorsqu'une affectation d'une variable entière est faite avec une valeur en partie droite inconnue ou un pointeur (cas de transtypage), les contraintes sur la variable en partie gauche sont supprimées.

- $v \in \mathcal{V}^{assign}$ **de la forme** $p = e$ **avec** $p \in \mathcal{V}ars^{ptr}$ **et** $e \in \textcircled{?} \cup \mathcal{V}ars^{\mathbb{Z}} \cup \mathbb{Z}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, p) \cup \{c\}, P[p \leftarrow \top])$$

Lorsqu'une affectation d'un pointeur est faite avec une valeur en partie droite inconnue ou une valeur entière (cas de transtypage), les contraintes sur la variable en partie gauche sont supprimées.

- $v \in \mathcal{V}^{assign}$ **de la forme** $w = e$ **avec** $w \in \overline{\mathcal{V}ars}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (C \cup \{c\}, P)$$

Si une affectation comporte en partie gauche un identifiant qui ne fait pas partie des variables prises en compte, la configuration n'est pas modifiée par le nœud.

$\underline{\mathcal{V}^{mem}}$

- $v \in \mathcal{V}^{mem}$ **de la forme** $*p = \alpha$ **avec** $p \in \mathcal{V}ars^{ptr}$**,** $\alpha \in \mathcal{V}ars^{\mathbb{Z}} \cup \mathbb{Z}$ **et** $P(p) = x \in \mathcal{V}ars^{\mathbb{Z}} \cap \mathcal{V}ars^{mem}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, x) \cup \{(x, =, \alpha)\} \cup \{c\}, P)$$

Une affectation via un pointeur p qui pointe sur une variable x entière connue permet d'ajouter une contrainte d'égalité entre la variable pointée et la partie droite. La fonction P n'est pas alterée.

- $v \in \mathcal{V}^{mem}$ **de la forme** $*p = e$ **avec** $p \in \mathcal{V}ars^{ptr}$**,** $e \in \mathcal{V}ars \cup \mathbb{Z}$**,** $P(p) = \top$ **et** $Oracle(p) \subseteq \mathcal{V}ars^{\mathbb{Z}}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (reset(C, Oracle(p)) \cup \{c\}, P)$$

Dans le cas où la variable pointée par p est inconnue, il faut supprimer de la configuration toutes les contraintes portant sur une variable possiblement pointée par p, d'après l'oracle d'alias de GCC. Ce cas est celui où p pointe sur des entiers.

- $v \in \mathcal{V}^{mem}$ **de la forme** $*p = r$ **avec** $p \in \mathcal{V}ars^{ptr}$**,** $r \in \mathcal{V}ars^{ptr}$ **et** $P(p) = q \in \mathcal{V}ars^{ptr} \cap \mathcal{V}ars^{mem}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, q) \cup \{(q, =, r)\} \cup \{c\}, P[q \leftarrow P(r)])$$

Si le pointeur p est un pointeur de pointeur, on ajoute la contrainte d'égalité et on modifie la fonction P en conséquence.

- $v \in \mathcal{V}^{mem}$ **de la forme** $*p = e$ **avec** $p \in \mathcal{V}ars^{ptr}$ **et** $P(p) = \top$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (reset(C, Oracle(p)) \cup \{c\}, \\ P[q \leftarrow \top \mid q \in Oracle(p) \cap \mathcal{V}ars^{ptr}])$$

Dans le cas où la variable pointée par p est inconnue et que p est un pointeur de pointeur, cela implique aussi de modifier la fonction P en plus de retirer les contraintes de C.

- $v \in \mathcal{V}^{mem}$ **de la forme** $*p = e$ **avec** $p \in \mathcal{V}ars^{ptr}$**,** $P(p) = x \in \mathcal{V}ars^{\mathbb{Z}}$ **et** $e \in \textcircled{?} \cup \mathcal{V}ars^{ptr}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, x) \cup \{c\}, P)$$

Si p pointe sur la variable x et que e est une valeur inconnue ou bien sur un pointeur (en cas de transtypage), on retire les contraintes sur x.

- $v \in \mathcal{V}^{mem}$ **de la forme** $*p = e$ **avec** $p \in \mathcal{V}ars^{ptr}$**,** $P(p) = \top$**,** $Oracle(p) \subseteq \mathcal{V}ars^{\mathbb{Z}}$ **et** $e \in \textcircled{?}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, Oracle(p)) \cup \{c\}, P)$$

Si p pointe sur une variable entière inconnue et que e est une valeur inconnue, on retire toutes les contraintes sur les variables pouvant être pointées par p, d'après l'oracle d'alias de GCC.

- $v \in \mathcal{V}^{mem}$ **de la forme** $*p = e$ **avec** $p \in \mathcal{V}ars^{ptr}$**,** $P(p) = q \in \mathcal{V}ars^{ptr}$ **et** $e \in \textcircled{?} \cup \mathcal{V}ars^{\mathbb{Z}} \cup \mathbb{Z}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, q) \cup \{c\}, P[q \leftarrow \top])$$

Si p pointe sur un pointeur q et que e est une valeur inconnue ou bien une valeur entière (en cas de transtypage), on retire les contraintes sur q et on modifie en conséquence la fonction P.

- $v \in \mathcal{V}^{mem}$ **de la forme** $*p = e$ **avec** $p \in \mathcal{V}ars^{ptr}$**,** $P(p) = \top$ **et** $e \in \textcircled{?} \cup \mathcal{V}ars^{\mathbb{Z}} \cup \mathbb{Z}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, \mathit{Oracle}(p)) \cup \{c\}, \\ P[q \leftarrow \top \mid q \in \mathit{Oracle}(p)])$$

Si p pointe sur un pointeur inconnu et que e est une valeur inconnue, on retire les contraintes sur tous les pointeurs sur lesquels p pourrait pointer et on modifie en conséquence la fonction P.

$\underline{\mathcal{V}^{\varphi}}$ Dans tous les cas suivants, $e_{path} \in \{e_1, \ldots, e_n\}$ est l'expression parmi les arguments du nœud ϕ correspondant au chemin analysé courant. La connaissance de quel argument est utilisé dans chaque chemin est fournie par GCC, c'est une propriété intrinsèque du chemin.

- $v \in \mathcal{V}^{\varphi}$ **de la forme** $x = \texttt{PHI}\langle e_1, \ldots, e_n\rangle$ **où** $x \in \mathcal{V}ars^{\mathbb{Z}}$ **et** $e_{path} \in \mathbb{Z} \cup \mathcal{V}ars^{\mathbb{Z}}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, x) \cup \{(x, =, e_{path})\} \cup \{c\}, P)$$

Un nœud ϕ se comporte essentiellement comme un nœud d'affectation.

- $v \in \mathcal{V}^{\varphi}$ **de la forme** $x = \texttt{PHI}\langle e_1, \ldots, e_n\rangle$ **où** $x \in \mathcal{V}ars^{\mathbb{Z}}$ **et** $e_{path} \in \textcircled{?} \cup \mathcal{V}ars^{ptr}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, x) \cup \{c\}, P)$$

- $v \in \mathcal{V}^{\varphi}$ **de la forme** $p = \texttt{PHI}\langle e_1, \ldots, e_n\rangle$ **où** $p \in \mathcal{V}ars^{ptr}$ **et** $e_{path} \in \mathcal{V}ars^{ptr}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, x) \cup \{\langle x, =, e_{path}\rangle\} \cup \{c\}, \\ P[x \leftarrow P(e_{path})])$$

- $v \in \mathcal{V}^{\varphi}$ **de la forme** $p = \texttt{PHI}\langle e_1, \ldots, e_n\rangle$ **où** $p \in \mathcal{V}ars^{ptr}$ **et** $e_{path} = \&y$ **avec** $y \in \mathcal{V}ars^{mem}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, x) \cup \{c\}, P[x \leftarrow y])$$

- $v \in \mathcal{V}^{\varphi}$ **de la forme** $p = \texttt{PHI}\langle e_1, \ldots, e_n\rangle$ **où** $p \in \mathcal{V}ars^{ptr}$ **et** $e_{path} = \textcircled{?} \cup \mathcal{V}ars^{\mathbb{Z}} \cup \mathbb{Z}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, x) \cup \{c\}, P[x \leftarrow \top])$$

$\underline{\mathcal{V}^{call}}$

– $v \in \mathcal{V}^{call}$ **de la forme** $x = f(e_1, \ldots, e_n)$ **avec** $x \in \mathcal{V}ars^{\mathbb{Z}}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, \{x\} \cup \mathcal{V}ars^{mem}) \cup \{c\}, \\ P[q \leftarrow \top \mid q \in \mathcal{V}ars^{mem} \cap \mathcal{V}ars^{ptr}])$$

Une fonction peut contenir toute sorte d'instructions. Par conséquent, il est impossible de prévoir les conséquences sur les valeurs des variables que l'appel d'une fonction entraîne. On efface par conséquent toutes les contraintes portant sur les variables vivant en mémoire, y compris les pointeurs. Les variables de $\mathcal{V}ars^{temp}$ ne sont pas impactées car elles sont propres à chaque fonction.

– $v \in \mathcal{V}^{call}$ **de la forme** $p = f(e_1, \ldots, e_n)$ **avec** $p \in \mathcal{V}ars^{ptr}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, \{p\} \cup \mathcal{V}ars^{mem}) \cup \{c\}, \\ P[q \leftarrow \top \mid q \in (\mathcal{V}ars^{mem} \cap \mathcal{V}ars^{ptr}) \cup p])$$

– $v \in \mathcal{V}^{call}$ **de la forme** $w = f(e_1, \ldots, e_n)$ **avec** $w \notin \mathcal{V}ars$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, \{w\} \cup \mathcal{V}ars^{mem}) \cup \{c\}, \\ P[q \leftarrow \top \mid q \in \mathcal{V}ars^{mem} \cap \mathcal{V}ars^{ptr}])$$

– $v \in \mathcal{V}^{call}$ **de la forme** $f(e_1, \ldots, e_n)$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (\text{reset}(C, \{w\} \cup \mathcal{V}ars^{mem}) \cup \{c\}, \\ P[q \leftarrow \top \mid q \in \mathcal{V}ars^{mem} \cap \mathcal{V}ars^{ptr}])$$

$\underline{\mathcal{V}^{join}}$

– $v \in \mathcal{V}^{join}$

$$(C, P) \overset{v,c}{\twoheadrightarrow} (C \cup \{c\}, P)$$

Les nœuds de jonction ne modifient pas la configuration courante.

En Coq, contrairement au cas de la sémantique concrète, nous pouvons définir complètement la relation.

```
Inductive AbstractSemantics : AbstractConfiguration → Nodes → Constraint
    → AbstractConfiguration → Prop :=
| asem_assignZ_to_Varz C P (x:Vars) (Hnptr_x: ¬pointer x) (z:Z) c:
  AbstractSemantics (C,P) (assignZ_to_Varz x Hnptr_x z) c
   (set_add Constraint_eq_dec (constraint1 x Eq z)
    (set_add Constraint_eq_dec c (reset C x)), P)
| asem_assignVarz_to_Varz C P (x:Vars) (Hnptr_x: ¬pointer x)
  (v:Vars) (Hnptr_v: ¬pointer v) c:
  AbstractSemantics (C,P) (assignVarz_to_Varz x Hnptr_x v Hnptr_v) c
   (set_add Constraint_eq_dec (constraint2 x Eq v)
    (set_add Constraint_eq_dec c (reset C x)), P)
| asem_assignPtr_to_Ptr C P (x:Vars) (Hptr_x: pointer x)
  (e:Vars) (Hptr_e: pointer e) c Hconsistency:
  AbstractSemantics (C,P) (assignPtr_to_Ptr x Hptr_x e Hptr_e Hconsistency) c
```

```
    (set_add Constraint_eq_dec (constraint2 x Eq e)
      (set_add Constraint_eq_dec c (reset C x)),
      (update_with_existing_element P x Hptr_x e Hptr_e Hconsistency))
| asem_assignAddr_to_Ptr C P (x:Vars) (Hptr_x: pointer x)
  (y:Vars) (Haddr_y: addressable y) c Hconsistency:
  AbstractSemantics (C,P) (assignAddr_to_Ptr x Hptr_x y Haddr_y Hconsistency) c
   (set_add Constraint_eq_dec c (reset C x),
    (update_with_address P x Hptr_x y Haddr_y Hconsistency))
| asem_assignDerefPtr_to_Varz C P
  (x:Vars) (Hnptr_x: ¬pointer x) (p:Vars) (Hptr_p: pointer p) c
  (v:Vars) (Haddr_v: addressable v)
  (Hp: (proj1_sig P) p Hptr_p = ptvar v Haddr_v) Hptp:
  AbstractSemantics (C,P) (assignDeref_to_Varz x Hnptr_x p Hptr_p Hptp) c
   (set_add Constraint_eq_dec (constraint2 x Eq v)
    (set_add Constraint_eq_dec c (reset C x)), P)
| asem_assignDerefUnknown_to_Varz C P
  (x:Vars) (Hnptr_x: ¬pointer x)
  (p:Vars) (Hptr_p: pointer p) c
  (Hp: (proj1_sig P) p Hptr_p = ptunknown) Horacle:
  AbstractSemantics (C,P) (assignDeref_to_Varz x Hnptr_x p Hptr_p Horacle) c
   (set_add Constraint_eq_dec c (reset C x), P)
| asem_assignDeref_to_Ptr C P (x:Vars) (Hptr_x: pointer x)
  (p:Vars) (Hptr_p: pointer p) c (v:Vars) (Haddr_v: addressable v)
  (Hp: (proj1_sig P) p Hptr_p = ptvar v Haddr_v)
  Hptp Hconsistency:
  AbstractSemantics (C,P) (assignDeref_to_Ptr x Hptr_x p Hptr_p Hptp
     Hconsistency) c
   (set_add Constraint_eq_dec (constraint2 x Eq v)
    (set_add Constraint_eq_dec c (reset C x)),
     (update_with_existing_element P x Hptr_x v
      (Hptp _ _ ((proj2_sig P) _ _ _ _ Hp))
      (Hconsistency _ _ _ ((proj2_sig P) _ _ _ _ Hp))))
| asem_assignOther_to_Varz C P (x:Vars) (Hnptr_x: ¬pointer x) (e:OtherExprs) c:
  AbstractSemantics (C,P) (assignOther_to_Varz x Hnptr_x e) c
   (set_add Constraint_eq_dec c (reset C x), P)
| asem_assignPtr_to_Varz C P (x:Vars) (Hnptr_x: ¬pointer x)
  (e:Vars) (Hptr_e: pointer e) c:
  AbstractSemantics (C,P) (assignPtr_to_Varz x Hnptr_x e Hptr_e) c
   (set_add Constraint_eq_dec c (reset C x), P)
| asem_assignDerefUnknown_to_Ptr C P (x:Vars) (Hptr_x: pointer x)
  (p:Vars) (Hptr_p: pointer p) c (Hp: (proj1_sig P) p Hptr_p = ptunknown)
  Hptp Hconsistency:
  AbstractSemantics (C,P)
   (assignDeref_to_Ptr x Hptr_x p Hptr_p Hptp Hconsistency) c
    (set_add Constraint_eq_dec c (reset C x),
     (update_with_top P x Hptr_x))
| asem_assignOther_to_Ptr C P (x:Vars) (Hptr_x: pointer x) (e:OtherExprs) c:
  AbstractSemantics (C,P) (assignOther_to_Ptr x Hptr_x e) c
   (set_add Constraint_eq_dec c (reset C x),
    (update_with_top P x Hptr_x))
| asem_assignZ_to_Ptr C P (x:Vars) (Hptr_x: pointer x) (e:Z) c:
  AbstractSemantics (C,P) (assignZ_to_Ptr x Hptr_x e) c
   (set_add Constraint_eq_dec c (reset C x),
    (update_with_top P x Hptr_x))
```

```
| asem_assignVarz_to_Ptr C P (x:Vars) (Hptr_x: pointer x)
  (e:Vars) (Hnptr_e: ¬pointer e) c:
  AbstractSemantics (C,P) (assignVarz_to_Ptr x Hptr_x e Hnptr_e) c
70  (set_add Constraint_eq_dec c (reset C x),
     (update_with_top P x Hptr_x))
| asem_assign_to_IgnoredVar C P (x:IgnoredVars) (e:Exprs) c:
  AbstractSemantics (C,P) (assign_to_IgnoredVar x e) c (set_add
    Constraint_eq_dec c C,P)

75 | asem_memZ_to_Ptr C P (p:Vars) (Hptr_p: pointer p) (z:Z) c
  (v:Vars) (Haddr_v: addressable v)
  (Hptp: (proj1_sig P) p Hptr_p = ptvar v Haddr_v) Hnonpointers:
  AbstractSemantics (C,P) (memZ_to_Ptr p Hptr_p z Hnonpointers) c
    (set_add Constraint_eq_dec (constraint1 v Eq z)
80   (set_add Constraint_eq_dec c (reset C v)), P)
| asem_memVarz_to_Ptr C P (p:Vars) (Hptr_p: pointer p)
  (e:Vars) (Hnptr_e: ¬ pointer e) c (v:Vars) (Haddr_v: addressable v)
  (Hptp: (proj1_sig P) p Hptr_p = ptvar v Haddr_v) Hnonpointers:
  AbstractSemantics (C,P) (memVarz_to_Ptr p Hptr_p e Hnptr_e Hnonpointers) c
85  (set_add Constraint_eq_dec (constraint2 v Eq e)
     (set_add Constraint_eq_dec c (reset C v)), P)
| asem_memZ_to_Unknown C P (p:Vars) (Hptr_p: pointer p) (z:Z) c
  (Hptp: (proj1_sig P) p Hptr_p = ptunknown) Hnonpointers:
  AbstractSemantics (C,P) (memZ_to_Ptr p Hptr_p z Hnonpointers) c
90  (set_add Constraint_eq_dec c (reset_all_pt C p Hptr_p), P)
| asem_memVarz_to_Unknown C P (p:Vars) (Hptr_p: pointer p)
  (e:Vars) (Hnptr_e: ¬ pointer e) c
  (Hptp: (proj1_sig P) p Hptr_p = ptunknown) Hnonpointers:
  AbstractSemantics (C,P) (memVarz_to_Ptr p Hptr_p e Hnptr_e Hnonpointers) c
95  (set_add Constraint_eq_dec c (reset_all_pt C p Hptr_p), P)
| asem_memPtr_to_Ptr C P
  (p:Vars) (Hptr_p: pointer p) (e:Vars) (Hptr_e: pointer e) c
  (v:Vars) (Haddr_v: addressable v)
  (Hptp: (proj1_sig P) p Hptr_p = ptvar v Haddr_v)
100 (Hpointers: pointers_to_pointers p Hptr_p) Hconsistency:
  AbstractSemantics (C,P)
  (memPtr_to_Ptr p Hptr_p e Hptr_e Hpointers Hconsistency) c
   (set_add Constraint_eq_dec (constraint2 v Eq e)
    (set_add Constraint_eq_dec c (reset C v)),
105  (update_with_existing_element P
     v (Hpointers _ _ ((proj2_sig P) _ _ _ _ Hptp))
     e Hptr_e
     (Hconsistency _ _ _ ((proj2_sig P) _ _ _ _ Hptp))))
| asem_memPtr_to_Unknown C P (p:Vars) (Hptr_p: pointer p)
110 (e:Vars) (Hptr_e: pointer e) c
  (Hptp: (proj1_sig P) p Hptr_p = ptunknown)
  (Hpointers: pointers_to_pointers p Hptr_p) Hconsistency:
  AbstractSemantics (C,P)
  (memPtr_to_Ptr p Hptr_p e Hptr_e Hpointers Hconsistency) c
115  (set_add Constraint_eq_dec c (reset_all_pt C p Hptr_p),
     (update_all_pt_with_top P p Hptr_p))

| asem_memUnknownVarz_to_Ptr C P (p:Vars) (Hptr_p: pointer p)
 (e:OtherExprs) c
```

```
120   (v:Vars) (Haddr_v: addressable v) (Hnptr_v: ¬pointer v)
      (Hptp: (proj1_sig P) p Hptr_p = ptvar v Haddr_v) :
       AbstractSemantics (C,P)
        (memOther_to_Ptr p Hptr_p e) c
          (set_add Constraint_eq_dec c (reset C v), P)
125 | asem_memUnknownVarz_to_Unknown C P (p:Vars) (Hptr_p: pointer p)
      (e:OtherExprs) c
      (Hptp: (proj1_sig P) p Hptr_p = ptunknown)
      (Hpointers: pointers_to_non_pointers p Hptr_p) :
       AbstractSemantics (C,P)
130     (memOther_to_Ptr p Hptr_p e) c
          (set_add Constraint_eq_dec c (reset_all_pt C p Hptr_p), P)

    | asem_memUnknownPtr_to_Ptr C P (p:Vars) (Hptr_p: pointer p)
      (e:OtherExprs) c
135   (q:Vars) (Haddr_q: addressable q) (Hptr_q: pointer q)
      (Hptp: (proj1_sig P) p Hptr_p = ptvar q Haddr_q) :
       AbstractSemantics (C,P)
        (memOther_to_Ptr p Hptr_p e) c
          (set_add Constraint_eq_dec c (reset C q),
140        update_with_top P q Hptr_q)
    | asem_memUnknownPtr_to_Unknown C P (p:Vars) (Hptr_p: pointer p)
      (e:OtherExprs) c
      (Hptp: (proj1_sig P) p Hptr_p = ptunknown) :
       AbstractSemantics (C,P)
145     (memOther_to_Ptr p Hptr_p e) c
          (set_add Constraint_eq_dec c (reset_all_pt C p Hptr_p),
           update_all_pt_with_top P p Hptr_p)

    | asem_mem_to_IgnoredVars C P (w:IgnoredVars) (e:Exprs) c :
150   AbstractSemantics (C,P)
        (mem_to_IgnoredVar w e) c (set_add Constraint_eq_dec c C,P)

    | asem_phiZ_to_Varz C P (x:Vars) (Hnptr_x: ¬pointer x) (Hnaddr_x: ¬addressable x)
      (z:Z) c:
155   AbstractSemantics (C,P) (phiZ_to_Varz x Hnptr_x Hnaddr_x z) c
        (set_add Constraint_eq_dec (constraint1 x Eq z)
          (set_add Constraint_eq_dec c (reset C x)), P)
    | asem_phiVarz_to_Varz C P
      (x:Vars) (Hnptr_x: ¬pointer x) (Hnaddr_x: ¬addressable x)
160   (v:Vars) (Hnptr_v: ¬pointer v) c:
      AbstractSemantics (C,P) (phiVarz_to_Varz x Hnptr_x Hnaddr_x v Hnptr_v) c
        (set_add Constraint_eq_dec (constraint2 x Eq v)
          (set_add Constraint_eq_dec c (reset C x)), P)
    | asem_phiOther_to_Varz C P
165   (x:Vars) (Hnptr_x: ¬pointer x) (Hnaddr_x: ¬addressable x)
      (e:OtherExprs) c:
      AbstractSemantics (C,P) (phiOther_to_Varz x Hnptr_x Hnaddr_x e) c
        (set_add Constraint_eq_dec c (reset C x), P)
    | asem_phiPtr_to_Ptr C P
170   (p:Vars) (Hptr_p: pointer p) (Hnaddr_p : ¬addressable p)
      (v:Vars) (Hptr_v: pointer v) c Hconsistency:
      AbstractSemantics (C,P)
       (phiPtr_to_Ptr p Hptr_p Hnaddr_p v Hptr_v Hconsistency) c
```

```
      (set_add Constraint_eq_dec (constraint2 p Eq v)
175    (set_add Constraint_eq_dec c (reset C p)),
       (update_with_existing_element P p Hptr_p v Hptr_v Hconsistency))
| asem_phiAddr_to_Ptr C P
  (p:Vars) (Hptr_p: pointer p) (Hnaddr_p: ¬ addressable p)
  (y:Vars) (Haddr_y: addressable y) c Hconsistency:
180  AbstractSemantics (C,P)
   (phiAddr_to_Ptr p Hptr_p Hnaddr_p y Haddr_y Hconsistency) c
    (set_add Constraint_eq_dec c (reset C p),
     (update_with_address P p Hptr_p y Haddr_y Hconsistency))
| asem_phiOther_to_Ptr C P
185  (p:Vars) (Hptr_p: pointer p) (Hnaddr_p: ¬addressable p)
  (e:OtherExprs) c:
  AbstractSemantics (C,P) (phiOther_to_Ptr p Hptr_p Hnaddr_p e) c
   (set_add Constraint_eq_dec c (reset C p),
    (update_with_top P p Hptr_p))
190
| asem_callVarz C P (x: Vars) (Hnptr_x: ¬pointer x) c:
  AbstractSemantics (C,P) (callVars x) c
   (set_add Constraint_eq_dec c (reset (reset_addr C) x),
    update_all_addr_with_top P)
195 | asem_callPtr C P (p: Vars) (Hptr_p: pointer p) c:
  AbstractSemantics (C,P) (callVars p) c
   (set_add Constraint_eq_dec c (reset (reset_addr C) p),
    update_all_addr_with_top (update_with_top P p Hptr_p))
| asem_callIgnoredVars C P (x:IgnoredVars) c:
200  AbstractSemantics (C,P) (callOther x) c
   (set_add Constraint_eq_dec c (reset_addr C),
    update_all_addr_with_top P)
| asem_callNoVars C P c:
  AbstractSemantics (C,P) call c
205  (set_add Constraint_eq_dec c (reset_addr C),
    update_all_addr_with_top P)
| asem_join C P c:
  AbstractSemantics (C,P) join c
    (set_add Constraint_eq_dec c C, P).
```

On étend naturellement ces sémantiques aux chemins. On écrit par exemple :

$$\theta \rightarrow_p^* \theta'$$

avec $p \in \mathcal{P}aths_{flows}$ pour signifier qu'il existe $v_1, v_2, \ldots, v_n \in \mathcal{V}$, $c_1, c_2, \ldots, c_n \in \mathcal{C}$, $\theta_1, \theta_2, \ldots, \theta_{n-1} \in \mathcal{K}$ tels que p est le chemin $(v_1, c_1, v_2, c_2, \ldots, v_{n-1}, c_n, v_n)$ et

$$\theta \xrightarrow{v_1,c_1} \theta_1 \xrightarrow{v_2,c_2} \theta_2 \ldots \theta_{n-1} \xrightarrow{v_n,c_n} \theta'.$$

En Coq, un chemin est une liste de couples ⟨nœud,contrainte⟩, la tête de liste étant le dernier nœud. Pour faciliter les preuves, nous définissons de manière artificielle la sémantique du chemin vide. On note également que contrairement à nos définitions « papier », en Coq, un chemin se termine par un arc sans destination, de manière assez surprenante. Ce n'est néanmoins pas une limitation puisqu'on peut toujours considérer l'existence d'un nœud de $\mathcal{V}^{join}$ terminal : les nœuds de cet ensemble ne font évoluer ni la sémantique abstraite, ni la sémantique concrète.

```
Definition Path : Type := list (Nodes * Constraint).
Inductive AbstractSemanticsPath : AbstractConfiguration → Path →
     AbstractConfiguration → Prop:=
 | AbstractSemanticsNil (k:AbstractConfiguration) :
   AbstractSemanticsPath k nil k
 | AbstractSemanticsCons (k k' k'': AbstractConfiguration)
    (p':Path) (IH: AbstractSemanticsPath k p' k')
    (v:Nodes) (c:Constraint) (Hlast: AbstractSemantics k' v c k'') :
   AbstractSemanticsPath k ((v,c) :: p') k''.

Inductive ConcreteSemanticsPath : MemoryState → Path → MemoryState → Prop
     :=
 | ConcreteSemanticsNil (theta:MemoryState) :
   ConcreteSemanticsPath theta nil theta
 | ConcreteSemanticsCons (theta theta' theta'':MemoryState)
    (p':Path) (IH: ConcreteSemanticsPath theta p' theta')
    (v:Nodes) (c:Constraint) (Hlast: ConcreteSemantics theta' v c theta'') :
   ConcreteSemanticsPath theta ((v,c) :: p') theta''.
```

Nous posons ici un lemme technique : nous affirmons que la relation de transition définissant la sémantique abstraite est complète à gauche, c'est-à-dire qu'il existe une règle pour chaque type de nœud.

Propriété 2 (Complétude à gauche de $\twoheadrightarrow$)**.**

$$\forall v \in \mathcal{V}\ \forall c \in \mathcal{C}\ \forall k \in \mathcal{K}\ \exists k' \in \mathcal{K}\ k \xrightarrow{v,c} k'$$

Démonstration. Nous avons prouvé ce lemme avec Coq, par analyse de cas sur tous les types de nœuds.

```
Theorem abstract_semantics_is_left_total :
∀C P n c,∃C' P', AbstractSemantics (C,P) n c (C',P').
```

□

5.4.4 Conclure qu'un chemin est impossible

Nous définissons une relation de satisfaisabilité entre configurations concrètes et abstraites. Intuitivement, une configuration concrète satisfait une configuration abstraite si chaque contrainte de la configuration abstraite est vérifiée par la valuation des variables dans la configuration concrète. Inversement, une configuration abstraite est insatisfaisable si aucune configuration concrète ne la satisfait ; typiquement, parce qu'elle contient des contraintes incompatibles entre elles. Un chemin est impossible s'il n'existe aucun moyen de produire une configuration abstraite satisfaisable en partant d'une configuration « vide », c'est-à-dire d'une configuration sans aucune contrainte, satisfaite par toutes les configurations concrètes. En effet, d'après nos définitions, cela revient à conclure qu'aucune valuation des variables ne peut être obtenue le long de ce chemin, donc qu'aucune exécution concrète ne peut le suivre.

Définition 11 (Satisfaisabilité)**.** Une configuration concrète $\theta = (\sigma, \gamma, \alpha) \in \Theta$ satisfait une configuration abstraite $k = (C, P) \in \mathcal{K}$, ce que l'on écrit $\theta \Vdash k$ si, et seulement si,

$$\begin{aligned} & \theta \vDash C \\ \wedge \quad & \forall p \in \mathcal{V}ars^{ptr}\ P(p) \neq \top \implies \alpha(P(p)) = \theta(p) \end{aligned}$$

La configuration k est insatisfaisable, ce que l'on note $\nVdash k$ si

$$\forall \theta \in \Theta\ \theta \nVdash k$$

En Coq, nous définissons la satisfaisabilité de la même manière.

```
Definition satisfiability_set_constraints theta C :=
  forall c, set_In c C -> satisfiability theta c.

Definition satisfiability_pointer_map theta (P:sig
    PointerMap_is_consistent) :=
  forall p Hptr_p v Haddr_v (Hptv: (proj1_sig P) p Hptr_p  =
    ptvar v Haddr_v),
  Valuation theta p = inr (alpha v Haddr_v).

Definition satisfiability_abstract_configuration theta k:=
  satisfiability_set_constraints theta (fst k) /\
  satisfiability_pointer_map theta (snd k).
```

Correction de l'analyse statique

Cette dernière définition nous permet d'exprimer la correction de notre analyse statique. En faisant l'hypothèse que les propriétés que nous avons supposées sur la sémantique concrète sont vérifiées, on peut prouver la proposition suivante.

Proposition 1 (Correction de l'analyse statique)**.** *La sémantique abstraite maintient la satisfaisabilité vis-à-vis de la configuration concrète correspondante.*

$$\begin{aligned} & \forall p \in \mathcal{P}aths_{flows}\ \forall \theta_1, \theta_2 \in \Theta\ \forall k_1, k_2 \in \mathcal{K} \\ & (\theta_1 \rightarrow_p^* \theta_2 \wedge k_1 \twoheadrightarrow_p^* k_2 \wedge \theta_1 \Vdash k_1) \implies \theta_2 \Vdash k_2 \end{aligned}$$

Intuitivement, cette proposition dit que si un chemin est possible, c'est-à-dire que sa sémantique concrète est définie, alors sa sémantique abstraite produit une configuration compatible avec la configuration concrète résultante.

Nous démontrons cette proposition par induction, en démontrant tout d'abord le cas de base dans le lemme ci-après.

Lemme 1 (Correction pour une transition)**.** $\forall \theta_1, \theta_2 \in \Theta\ \forall k_1, k_2 \in \mathcal{K}\ \forall v \in \mathcal{V}\ \forall c \in \mathcal{C}\ (\theta_1 \xrightarrow{v,c} \theta_2 \wedge k_1 \xrightarrow{v,c}\!\!\!\!\!\rightarrow\ k_2 \wedge \theta_1 \Vdash k_1) \Rightarrow \theta_2 \Vdash k_2$

Démonstration. Cette preuve a elle aussi été réalisée en Coq, par induction sur la définition de l'analyse statique (dont on a déjà prouvé qu'elle couvrait tous les types de nœuds).

```
Lemma restricted_soundness :∀(v:Nodes) (c:Constraint)
 (k k':AbstractConfiguration) (theta theta':MemoryState),
 (AbstractSemantics k v c k') → (ConcreteSemantics theta v c theta') →
 satisfiability_abstract_configuration theta k →
 satisfiability_abstract_configuration theta' k'.
```

□

On peut à présent prouver la proposition 1.

Démonstration. Nous faisons une simple induction sur la longueur de p, en utilisant le lemme 1.

```
Theorem soundness :∀(p:Path)
 (k k':AbstractConfiguration) (theta theta':MemoryState),
 (AbstractSemanticsPath k p k') → (ConcreteSemanticsPath theta p theta') →
 satisfiability_abstract_configuration theta k →
 satisfiability_abstract_configuration theta' k'.
```

□

Prouver l'impossibilité d'un chemin

Pour prouver qu'un chemin est impossible, on peut donc :

1. démarrer avec une configuration abstraite vide ;
2. progresser le long du chemin en faisant évoluer la configuration d'après les règles de la sémantique présentées en section 5.4.3 ;
3. s'arrêter lorsque la configuration est insatisfaisable et déclarer le chemin impossible ;
4. ou atteindre la fin du chemin et déclarer le chemin possible.

Formellement, on définit l'ensemble $\mathbf{I}$ des chemins impossibles, et l'ensemble $\mathbf{E}$ des chemins exécutables, comme suit :

Définition 12 (Chemins impossibles et chemins exécutables)**.**

$$\mathbf{I} = \{p \in \mathcal{P}aths \mid \forall k_1, k_2 \in \mathcal{K}\ k_1 \twoheadrightarrow_p^* k_2 \implies \nVdash k_2\}$$
$$\mathbf{E} = \{p \in \mathcal{P}aths \mid \exists \theta_1, \theta_2 \in \Theta\ \theta_1 \rightarrow^* \theta_2\}$$

On établit la proposition suivante.

Proposition 2 (Les chemins détectés comme impossibles le sont vraiment)**.** *Si un chemin est détecté comme impossible par l'analyse statique, alors aucune exécution concrète ne peut l'emprunter.*

$$\mathbf{E} \cap \mathbf{I} = \emptyset$$

Démonstration. Supposons qu'il existe $p \in \mathbf{I} \cap \mathbf{E}$. Comme $p \in \mathbf{E}$, il existe $\theta_1, \theta_2 \in \Theta$ tels que $\theta_1 \rightarrow_p^* \theta_2$. Soit alors $k_1 \in \mathcal{K}$ une configuration abstraite telle que $\theta_1 \Vdash k_1$ (une telle configuration existe toujours car il suffit de considérer (C, P) où $C = \{\texttt{true}\}$ et $P = x \mapsto \top$ qui satisfait clairement cette propriété). Comme $\twoheadrightarrow$ est une relation totale

à gauche, il existe donc une configuration abstraite $k_2 \in \mathcal{K}$ telle que $k_1 \twoheadrightarrow_p^* k_2$. Par conséquent, par la proposition de correction établie plus haut, on a $\theta_2 \Vdash k_2$. Cependant, comme $p \in \mathbf{I}$, on a par hypothèse $\not\Vdash k_2$, ce qui donne une contradiction. □

Finalement, d'après cette proposition, si l'on montre que $\mathbf{P} \subseteq \mathbf{I}$, alors on prouve $\mathbf{P} \cap \mathbf{E} = \emptyset$. Les chemins qui contiennent les instructions générant un flux d'information tout en esquivant les crochets LSM (l'ensemble $\mathbf{P}$) ne sont pas empruntables lors d'exécutions réelles. Par conséquent, ils ne permettent pas d'esquiver le contrôle de flux d'information.

5.4.5 Gestion des boucles

La proposition 1 établie à la section précédente montre que l'on peut considérer tous les chemins de longueur finie. Les chemins infinis ne sont pas pertinents dans notre étude puisqu'un appel système ne terminant jamais son exécution est synonyme d'une erreur de programmation du noyau et ne présente pas d'intérêt en termes de contrôle de flux.

Cependant, si le graphe contient des cycles, correspondant à des boucles dans le code, l'ensemble $\mathbf{P}$ des chemins de longueur arbitrairement grande peut être infini. Heureusement, le compilateur applique quelques restrictions à la représentation des cycles dans les graphes. En premier lieu, toutes les boucles possèdent un unique nœud de rebouclage qui est un nœud de jonction possédant exactement deux arcs : un provenant d'avant la boucle et un provenant de l'intérieur de celle-ci (ce qui forme ainsi un cycle dans le graphe). Ordinairement, il s'agit du nœud d'entrée dans la boucle mais il est possible en C de sauter à l'intérieur d'une boucle et ces cas particuliers existent également en GIMPLE. D'autre part, deux boucles imbriquées sont toujours différenciées, elles ne partagent jamais le même nœud de rebouclage. Nous faisons l'hypothèse que ces propriétés sont vérifiées. Dans la pratique, elles sont garanties par GCC.

Définition 13 (Boucle)**.** Une boucle l est définie par la donnée de :

- $\mathcal{V}_l = \{v_l, v_1, v_2, \ldots, v_n\} \subseteq \mathcal{V}$;
- $v_l \in \mathcal{V}_l \cap \mathcal{V}^{join}$;
- et $\mathcal{T}_l = (v_l, c_1, v_1), (v_1, c_2, v_2), \ldots, (v_n, c_n, v_l) \in \mathcal{E}^+$ dans le graphe.

Les propriétés suivantes sont respectées :

- v_l est l'unique nœud de $\mathcal{V}_l$ à avoir un prédecesseur dans $\mathcal{V} \setminus \mathcal{V}_l$. Il est nommé nœud de rebouclage de la boucle.
- v_l n'est le nœud de rebouclage d'aucune autre boucle du graphe.

Nous définissons une relation d'équivalence entre les chemins de $\mathbf{P}$. Nous posons que deux chemins sont équivalents s'ils sont identiques aux cycles près, c'est-à-dire si retirer tous les cycles de chacun des deux chemins donne le même chemin acyclique.

Définition 14 (Forme normale d'un chemin et relation d'équivalence)**.** Soit un chemin $p \in \mathcal{P}aths$, la forme normale de p notée $\hat{p}$ est le chemin de $\mathbf{P}$ où tous les cycles, c'est-à-dire les sous-chemins d'un nœud $v \in \mathcal{V}$ à ce même nœud v sont supprimés.

On dit de deux chemins $p_1, p_2 \in \mathcal{P}aths$ qu'ils sont équivalents, ce que l'on note $p_1 \equiv p_2$ si, et seulement si, $\hat{p_1} = \hat{p_2}$.

Il est clair que $\equiv$ est effectivement une relation d'équivalence car l'égalité est une relation réflexive, symmétrique et transitive. Nous partitionnons $\mathcal{P}aths$ d'après cette relation d'équivalence. On montre aisément qu'il existe une unique forme normale dans chaque classe d'équivalence.

Configuration résultat des boucles

Pour chaque boucle, nous calculons une configuration abstraite qui ne dépend pas du nombre d'itérations en retirant de la configuration de début de boucle toutes les contraintes portant sur les variables modifiées à l'intérieur de la boucle (cela peut correspondre à toutes les variables de $\mathcal{V}ars^{mem}$ si un appel de fonction est présent entre autres). Nous appelons cette configuration « la configuration résultat de la boucle ».

Définition 15 (Configuration résultat d'une boucle)**.** Soient une boucle $l = (\mathcal{V}_l, v_l, \mathcal{T}_l)$ telle que $\mathcal{T}_l = (v_l, c_1, v_1), (v_1, c_2, v_2), \ldots, (v_n, c_n, v_l)$ et une configuration $k \in \mathcal{K}$. La configuration résultat de l à partir de k est notée k^l. On la calcule en retirant de k toutes les contraintes portant sur les variables modifiées à l'intérieur de la boucle.

Formellement, nous définissons une fonction $\mathrm{purge} : \mathcal{K} \times \mathcal{V} \to \mathcal{K}$

- $v \in \mathcal{V}^{assign}$ de la forme $x = e$ avec $x \in \mathcal{V}ars^{\mathbb{Z}}$

$$\mathrm{purge}((C, P), v) = (\mathrm{reset}(C, x), P)$$

- $v \in \mathcal{V}^{assign}$ de la forme $x = e$ avec $x \in \mathcal{V}ars^{ptr}$

$$\mathrm{purge}((C, P), v) = (\mathrm{reset}(C, x), P[x \leftarrow \top])$$

- $v \in \mathcal{V}^{assign}$ de la forme $x = e$ avec $x \in \overline{\mathcal{V}ars}$

$$\mathrm{purge}((C, P), v) = (C, P)$$

- $v \in \mathcal{V}^{mem}$ de la forme $*p = e$ avec $P(p) \in \mathcal{V}ars^{\mathbb{Z}}$

$$\mathrm{purge}((C, P), v) = (\mathrm{reset}(C, P(p)), P)$$

- $v \in \mathcal{V}^{mem}$ de la forme $*p = e$ avec $P(p) \in \mathcal{V}ars^{ptr}$

$$\mathrm{purge}((C, P), v) = (\mathrm{reset}(C, P(p)), P[p \leftarrow \top])$$

- $v \in \mathcal{V}^{mem}$ de la forme $*p = e$ avec $P(p) = \top$

$$\mathrm{purge}((C, P), v) = (\mathrm{reset}(C, \mathcal{O}(p)), P[q \leftarrow \top \mid \mathcal{O}(P) \cap \mathcal{V}ars^{ptr}])$$

- $v \in \mathcal{V}^{\varphi}$ de la forme $x = \texttt{PHI}\langle e_1, \ldots, e_n \rangle$ avec $x \in \mathcal{V}ars^{\mathbb{Z}}$

$$\mathrm{purge}((C, P), v) = (\mathrm{reset}(C, x), P)$$

– $v \in \mathcal{V}^{\varphi}$ de la forme $x = \texttt{PHI}\langle e_1, \ldots, e_n \rangle$ avec $x \in \mathcal{V}ars^{ptr}$

$$\text{purge}((C, P), v) = (\text{reset}(C, x), P[x \leftarrow \top])$$

– $v \in \mathcal{V}^{call}$ de la forme $f()$

$$\text{purge}((C, P), v) = (\text{reset}(C, \mathcal{V}ars^{mem}), P[q \leftarrow \top \mid q \in \mathcal{V}ars^{mem} \cap \mathcal{V}ars^{ptr}])$$

– $v \in \mathcal{V}^{call}$ de la forme $f(e_1, \ldots, e_n)$

$$\text{purge}((C, P), v) = (\text{reset}(C, \mathcal{V}ars^{mem}), P[q \leftarrow \top \mid q \in \mathcal{V}ars^{mem} \cap \mathcal{V}ars^{ptr}])$$

– $v \in \mathcal{V}^{call}$ de la forme $x = f()$ avec $x \in \mathcal{V}ars$

$$\text{purge}((C, P), v) = (\text{reset}(C, \mathcal{V}ars^{mem} \cup \{x\}), P[q \leftarrow \top \mid q \in (\mathcal{V}ars^{mem} \cap \mathcal{V}ars^{ptr}) \cup \{x\}])$$

– $v \in \mathcal{V}^{call}$ de la forme $x = f(e_1, \ldots, e_n)$

$$\text{purge}((C, P), v) = (\text{reset}(C, \mathcal{V}ars^{mem}), P[q \leftarrow \top \mid q \in (\mathcal{V}ars^{mem} \cap \mathcal{V}ars^{ptr}) \cup \{x\}])$$

La configuration résultat de la boucle l à partir de la configuration k est définie comme :

$$k^l = \text{purge}(\text{purge}(\ldots(\text{purge}(k, v_n), \ldots), v_1), v_l)$$

Nous démontrons que cette manière de procéder est correcte en montrant que la configuration abstraite obtenue est satisfaite par au moins les mêmes configurations concrètes que toute configuration qui serait obtenue par l'application des règles de la sémantique abstraite le long d'un chemin de la boucle l (c'est-à-dire un chemin constitué d'une nombre d'itérations arbitrairement grand mais fini du cycle constituant l). La preuve repose sur une relation d'ordre entre les configurations abstraites et la propriété que cet ordre est compatible avec la satisfaisabilité.

Définition 16 (Relation d'ordre $\preceq$ sur les configurations abstraites)**.** On définit la relation d'ordre $\preceq \subseteq \mathcal{K} \times \mathcal{K}$ sur les configurations abstraites par :

$$(C, P) \preceq (C', P') \Leftrightarrow C \subseteq C' \wedge (\forall p \in \mathcal{V}ars^{ptr}\ P(p) = \top \vee P(p) = P'(p))$$

Proposition 3 (Compatibilité de $\preceq$ avec $\Vdash$)**.**

$$\forall \theta \in \Theta\ \forall k, k' \in \mathcal{K}\ k \preceq k' \Rightarrow (\theta \Vdash k' \Rightarrow \theta \Vdash k)$$

Démonstration. Supposons que l'on ait $k \preceq k'$, $\theta \Vdash k'$ mais $\theta \nVdash k$. Alors, posant $\theta = (\gamma, \sigma, \alpha)$, $k = (C, P)$ et $k' = (C', P')$, on sait que

$$(\exists c \in C\ \theta \nvDash c) \vee (\exists p \in \mathcal{V}ars^{ptr}\ P(p) \neq \top \wedge \alpha(p) \neq \theta(p))$$

Par analyse des cas :

— $\exists c \in C\ \theta \not\models c$. Si $c \in C$, alors $c \in C'$ car $C \subseteq C'$ par hypothèse. Mais, on sait que $\theta \Vdash C'$, donc $\theta \models c$, d'où une contradiction.

— $\exists p \in \mathcal{V}ars^{ptr}\ P(p) \neq \top \wedge \alpha(x) \neq \theta(p)$. Si $P(p) \neq \top$ alors $P'(p) = P(p)$ par hypothèse. Mais on sait que $\forall p \in \mathcal{V}ars^{ptr}\ P'(p) \neq \top \Rightarrow \alpha(p) = \theta(p)$, donc on a $P(p) \neq \top \wedge \alpha(p) = \theta(p)$, en contradiction avec les hypothèses.

□

Nous démontrons à présent un lemme indiquant que la configuration résultat de la boucle l est plus petite que toute configuration abstraite calculée à partir d'un chemin de l.

Lemme 2 (La configuration résultat est plus petite que toute configuration calculée sur un chemin de l)**.** *Soit une boucle* $l = (\mathcal{V}_l, v_l, \mathcal{T}_l)$.

$$\forall (v, c, v') \in \mathcal{T}_l\ \forall k, k' \in \mathcal{K}\ k \xrightarrow{v,c}\!\!\!\!\rightarrow k' \Rightarrow k^l \preceq k'$$

Démonstration. On suppose que $k = (C, P)$, $k^l = (C^l, P^l)$ et $k' = (C', P')$. On doit alors prouver :

1. $C^l \subseteq C'$
2. $\forall p \in \mathcal{V}ars^{ptr}\ P^l(p) = \top \vee P^l(p) = P'(p)$

1. $C \subseteq C'$. Par analyse des cas selon la nature de v, et par la définition des règles sémantiques, il est clair que les seules contraintes de C qui ne soient pas dans C' sont des constraintes retirées par la fonction reset. Ces contraintes portent donc sur la variable assignée dans un nœud d'affectation, ou bien la variable $P(p)$ ou une variable de $\mathcal{O}(p)$ dans le cas d'une affectation à travers un pointeur p, ou encore $\mathcal{V}ars^{mem}$ dans le cas d'un appel de fonction. Par la définition de k^l, ces contraintes de C ne font pas partie de C^l. Comme de plus $C^l \subseteq C$, on conclut que $C^l \subseteq C'$.
2. $\forall p \in \mathcal{V}ars^{ptr}\ P^l(p) = \top \vee P^l(p) = P'(p)$ Tous les pointeurs p tels que $P(p) \neq \top \wedge P'(p) \neq P(p)$ sont des pointeurs assignés directement ou des pointeurs dans $\mathcal{V}ars^{mem}$ assignés à travers un pointeur. Dans tous les cas, par définition de k^l, on a $P^l(p) = \top$ donc on peut conclure que $\forall p \in \mathcal{V}ars^{ptr}\ P^l(p) = \top \vee P^l(p) = P'(p)$.

□

On peut à présent prouver une version du lemme précédent étendu à un chemin entier.

Proposition 4. *Soient une boucle* $l = (\mathcal{V}_l, v_l, \mathcal{T}_l)$ *et* p *un chemin de* $\mathcal{T}_l$*. On a :*

$$\forall k, k' \in \mathcal{K}\ \forall \theta \in \Theta\ (k \twoheadrightarrow_p^* k' \wedge \theta \Vdash k') \Rightarrow \theta \Vdash k^l$$

Démonstration. Comme $\preceq$ est une relation d'ordre, elle est transitive et par induction sur la longueur du chemin p, on peut conclure d'après le lemme 2 que $k \twoheadrightarrow_p^* k' \Rightarrow k^l \preceq k'$. Par la proposition 3, on a donc $\forall \theta \in \Theta\ \theta \Vdash k' \Rightarrow \theta \Vdash k^l$. □

Conclusion sur l'analyse des chemins

Il ressort de la dernière proposition qu'il suffit d'analyser la forme normale de chaque classe d'équivalence. L'ensemble des formes normales est l'ensemble (fini) des chemins acycliques du graphes. Pour l'analyse, nous nous restreignons à calculer le sous-ensembles des chemins acycliques de $\mathbf{P}$. Nous montrons que si la forme normale correspond à un chemin impossible alors tous les chemins de l'ensemble sont impossibles. Nous analysons ensuite les formes normales et concluons ainsi sur l'impossibilité de chacun des chemins de $\mathbf{P}$.

Considérons $S \in \mathbf{P}_{\setminus\equiv}$ une classe d'équivalence selon $\equiv$ et p un chemin de S. Nous voulons prouver que $\hat{p} \in \mathbf{I}$ implique $p \in \mathbf{I}$.

Nous montrons dans un premier temps que les règles de la sémantique abstraite préserve l'ordre sur les configurations abstraites.

Lemme 3 (Préservation)**.**

$$\forall k_1, k_1', k_2, k_2' \in \mathcal{K}\ \forall v \in \mathcal{V}\ \forall c \in \mathcal{C}\ \left(\begin{cases} k_1 \xrightarrow{v,c}\!\!\!\!\rightarrow k_2 \\ k_1' \xrightarrow{v,c}\!\!\!\!\rightarrow k_2' \end{cases} \wedge k_1' \preceq k_1 \right) \Rightarrow k_2' \preceq k_2$$

Démonstration. Supposons que $k_1 = (C_1, P_1)$, $k_2 = (C_2, P_2)$, $k_1' = (C_1', P_1')$, $k_2' = (C_2', P_2')$, et $k_1' \preceq k_1$.

Soit $q \in \mathcal{V}ars^{ptr}$ un pointeur. Par la définition des règles sémantiques, on moins un des cas suivants s'applique :

1. **Cas** $\begin{cases} P_2'(q) = P_1'(q) \\ P_2(q) = P_1(q) \end{cases}$.
   C'est le cas où (v, c) ne change pas l'information de la variable sur laquelle q pointe. Comme $k_1' \preceq k_1'$, on a $P_1'(q) = \top$ ou $P_1'(q) = P_1(q)$, ce qui donne $P_2'(q) = \top$ ou $P_2'(q) = P_2(q)$.

2. **Cas** $\begin{cases} P_2'(q) = \top \\ P_2(q) = \top \end{cases}$.
   C'est le cas où (v, c) change l'information à propos de la variable sur laquelle q pointe à $\top$. Dans ce cas, on a trivialement $P_2'(q) = \top$ ou $P_2'(q) = P_2(q)$.

3. **Cas** $\exists q' \in \mathcal{V}ars^{ptr}$ $\begin{cases} P_2'(q) = P_1'(q') \\ P_2(q) = P_1(q') \end{cases}$.
   C'est le cas où la transition (v, c) change l'information à propos de la variable sur laquelle q pointe à la destination d'un autre pointeur q'. Comme $k_1' \preceq k_1'$, on a $P_1'(q') = \top$ ou $P_1'(q') = P_1(q')$, ce qui donne $P_2'(q) = \top$ ou $P_2'(q) = P_2(q)$.

4. **Cas** $\exists x \in \mathcal{V}ars^{mem}$ $\begin{cases} P_2'(q) = x \\ P_2(q) = x \end{cases}$.
   C'est le cas où la transition (v, c) change l'information à propos de la variable sur laquelle q pointe à une variable x. Dans ce cas, on a trivialement $P_2'(q) = \top$ ou $P_2'(q) = P_2(q) = v$.

Soit $C_r, C_a \subseteq \mathcal{C}$ les ensembles de contraintes respectivement retirés et ajoutés à C_1 et C_1' par la transition (v, c) (d'après les règles de la sémantique abstraite). On a $\begin{cases} C_2 = (C_1 \setminus C_r) \cup C_a \\ C_2' = (C_1' \setminus C_r) \cup C_a \end{cases}$. Cela donne trivialement $C_1' \subseteq C_2' \implies C_1 \subseteq C_2$.

On a donc prouvé que $C'_2 \subseteq C_2$ et $\forall q \in \mathcal{V}ars^{ptr}\ (P'_2(q) = \top \vee P'_2(q) = P_2(q))$, ce qui revient à écrire $k'_2 \preceq k_2$. □

Nous étendons le lemme précédent aux chemins.

Lemme 4 (Préservation de l'ordre le long d'un chemin)**.**

$$\forall k_1, k'_1, k_2, k'_2 \in \mathcal{K}\ \forall p \in \mathbf{P}\ \left(\begin{cases} k_1 \twoheadrightarrow_p^* k_2 \\ k'_1 \twoheadrightarrow_p^* k'_2 \end{cases} \wedge k'_1 \preceq k_1\right) \Rightarrow k'_2 \preceq k_2$$

Démonstration. Par induction sur la longueur du chemin p (un chemin de longueur $n+1$ est un chemin de longueur n suivi d'une dernière transition (v, c)). Comme $\preceq$ est une relation d'ordre, elle est transitive, ce qui nous donne le résultat trivialement. □

Le lemme 2, p. 120 donne un résultat similaire à propos des boucles.
On peut donc finalement prouver la proposition suivante :

Proposition 5 (Analyser la forme normale d'une classe d'équivalence suffit à conclure sur l'impossibilité de la classe entière)**.** *Étant donné $S \subseteq \mathbf{P}$ une classe d'équivalence selon $\equiv$. et $[s] \in S$ la forme normale de S, on a $[s] \in \mathbf{I} \Rightarrow S \subseteq \mathbf{I}$.*

Démonstration. Considérons un chemin $p \in S$. Par définition $\hat{p} = [s]$. Supposons que $[s] \in \mathbf{I}$ mais $p \notin \mathbf{I}$. On peut alors supposer qu'il existe $k_1, k_2 \in \mathcal{K}$ $k_1 \twoheadrightarrow_p^* k_2$ tels que $\Vdash k_2$. p est une composition de sous-chemins, certains étant des cycles. Par définition, $[s]$ est le même chemin sans les cycles. Par induction sur la longueur de la séquence de sous-chemins de p et d'après le lemme 4 pour les sous-chemins n'étant pas des boucles, et le lemme 2 pour les sous-chemins qui en sont, et par transitivité de $\preceq$, il existe $k'_2 \in \mathcal{K}$ tel que $k_1 \twoheadrightarrow_{[s]}^* k'_2$ et $k'_2 \preceq k_2$. Par la proposition 3, il s'ensuit $\Vdash [s]$, en contradiction avec l'hypothèse. □

5.5 Implémentation

Depuis la version 4.3, le compilateur GCC supporte l'addition de passes de compilation supplémentaires dans sa chaîne de traitements sous la forme de greffons pouvant être chargés dynamiquement. Les greffons chargés peuvent bénéficier de toutes les représentations internes et toutes les fonctions implémentées par GCC. Normalement, ces greffons sont utilisés pour ajouter des optimisations mais ils peuvent également servir à l'implémentations d'analyses statiques. Quelques greffons ont notamment été intégrés aux sources du noyau dans les dernières versions.

Nous avons implémenté deux greffons pour réaliser l'analyse statique décrite dans cette section : *Kayrebt::Callgraphs*, déjà présenté dans le chapitre 4, et *Kayrebt::PathExaminer2*. Le premier est très simple et permet uniquement d'extraire du code un graphe d'appels de fonction du noyau. Ce graphe d'appels est une base de données indiquant pour chaque fonction, quelles sont les fonctions qui sont appelées dans son code, sans tenir compte des conditions ou de l'ordre de ces appels. Le graphe d'appels nous a permis d'identifier de manière automatique quels sont les crochets LSM atteignables depuis chaque appel système, ce qui nous a permis de forcer correctement leur *inlining*.

Le greffon *Kayrebt::PathExaminer2* est celui qui implémente réellement l'analyse. Nous n'extrayons pas du code des graphes comme ceux présentés en figure 5.1 ; en réalité, nous travaillons directement sur la représentation interne de GCC des graphes

de flots de contrôle. Ceci évite une transformation qui pourrait être source d'erreurs d'interprétation du code. De plus, on peut ainsi bénéficier d'autres structures de données du compilateur comme l'oracle d'alias. L'analyse est faite indépendamment sur chaque appel système et à l'intérieur de chaque appel système, sur chaque instruction générant un flux. Les points où des flux sont générés doivent être annotés à la main avec un pseudo-appel de fonction géré par notre greffon. Toutes les fonctions intermédiaires entre l'appel système et les fonctions où se trouvent les crochets LSM et les points de génération de flux doivent être marquées par un attribut spécifique à GCC afin de forcer leur *inlining*.

L'exécution de l'analyse démarre à l'entrée de l'appel système. Le code se contente essentiellement d'effectuer un parcours en largeur du graphe de flot de contrôle afin d'explorer tous les chemins méthodiquement. Au fur et à mesure de l'exploration, une configuration abstraite, initialement vide, est mise à jour avec les contraintes des nœuds et des arcs. À chaque modification de la configuration, sa satisfaisabilité est vérifiée. Nous utilisons Yices [23], un SMT-solveur équipé de la théorie de l'arithmétique, pour vérifier la satisfaisabilité des ensembles de contraintes. Si la configuration est insatisfaisable, alors le chemin est abandonné. Lorsqu'un branchement conditionnel est rencontré, avec plusieurs arcs sortants, la configuration est dupliquée et l'analyse se poursuit indépendamment sur chacun des chemins. Si une analyse se poursuit jusqu'à atteindre la fin du chemin (le retour de la fonction) avec une configuration satisfaisable, alors ce chemin ne peut pas être prouvé impossible. Il est affiché sur la sortie de l'outil pour permettre une vérification manuelle plus poussée.

5.6 Résultats

Nous avons initialement développé l'analyse pour le noyau officiel 4.3, compilé pour l'architecture x86_64 avec les options de configuration par défaut, mais nous l'avons par la suite reproduite sur le noyau 4.7. En réalité, l'analyse est indépendante de l'architecture et la version du noyau mais elle doit être conduite indépendamment dans chaque cas pour pouvoir tirer des conclusions utiles. En effet, selon l'architecture, la version et les options de configurations, les appels systèmes et les moyens de produire des flux d'information varient.

Nous avons cloné le dépôt de code officiel du noyau Linux puis nous avons préparé une branche pour chaque appel système dans laquelle nous avons placé les annotations correspondantes pour lancer l'analyse. Les résultats de cette dernière sont listés dans le tableau 5.2.

Dans la majorité des cas, le résultat de l'analyse est clair : soit il n'existe aucun chemin ne passant pas par un crochet LSM, soit tous ces chemins sont impossibles. Dans tous ces cas, on peut donc conclure que les crochets LSM sont placés de manière appropriée.

Certains appels système que nous avons identifiés comme provoquant des flux d'information ne possèdent pas de crochets LSM. C'est le cas de `tee`, provoquant un flux d'information de tube à tube, ainsi que de `mq_timedreceive` et `mq_timedsend` qui respectivement récupère et insère des messages dans des files de messages POSIX.

5.6.1 `mq_timedsend` et `mq_timedreceive`

Contrairement aux appels système correspondant des files de messages System V, `msgsnd` et `msgrcv`, les appels `mq_timedsend` et `mq_timedreceive` n'ont pas de cro-

TABLE 5.2 – Résultats de l'analyse statique

Appel système	Résultat	Détails
Flux discrets		
`read`	✓	Tous les chemins de **P** sont impossibles
`readv`	✓	Tous les chemins de **P** sont impossibles
`preadv`	✓	Tous les chemins de **P** sont impossibles
`pread64`	✓	Tous les chemins de **P** sont impossibles
`write`	✓	Tous les chemins de **P** sont impossibles
`writev`	✓	Tous les chemins de **P** sont impossibles
`pwritev`	✓	Tous les chemins de **P** sont impossibles
`pwrite64`	✓	Tous les chemins de **P** sont impossibles
`sendfile`	✓	Tous les chemins de **P** sont impossibles
`sendfile64`	✓	Tous les chemins de **P** sont impossibles
`splice`	∼	Pas de crochet pour le flux de tube à tube
		Tous les autres chemins sont impossibles
`tee`	×	Pas de crochet LSM
`vmsplice`	∼	Un chemin est possible
`recv`	✓	L'ensemble **P** est vide
`recvmsg`	✓	L'ensemble **P** est vide
`recvmmsg`	∼	Un chemin est possible
`recvfrom`	✓	L'ensemble **P** est vide
`send`	✓	L'ensemble **P** est vide
`sendmsg`	✓	L'ensemble **P** est vide
`sendmmsg`	∼	Un chemin est possible
`sendto`	✓	L'ensemble **P** est vide
`process_vm_readv`	✓	Des chemins sont possibles mais voir plus bas
`process_vm_writev`	✓	Des chemins sont possibles mais voir plus bas
`migrate_pages`	✓	L'ensemble **P** est vide
`move_pages`	✓	L'ensemble **P** est vide
`fork`	✓	L'ensemble **P** est vide
`vfork`	✓	L'ensemble **P** est vide
`clone`	✓	L'ensemble **P** est vide
`execve`	✓	L'ensemble **P** est vide
`execveat`	✓	L'ensemble **P** est vide
`msgrcv`	✓	Tous les chemins de **P** sont impossibles
`msgsnd`	✓	L'ensemble **P** est vide
`mq_timedreceive`	×	Pas de crochet LSM
`mq_timedsend`	×	Pas de crochet LSM

Suite de la TABLE 5.2. Résultats de l'analyse statique

Flux continus		
`shmat`	✓	L'ensemble **P** est vide
`mmap_pgoff`	✓	L'ensemble **P** est vide
`mmap`	✓	L'ensemble **P** est vide
`ptrace`	✓	Des chemins sont possibles

chets LSM. Nous avons interrogé les développeurs des modules LSM sur leur liste de diffusion sur les raisons de cette absence. Stephen SMALLEY, développeur et mainteneur de SELinux, a fourni les éléments d'explication suivants [84] :

— Ces appels système ont été ajoutés au noyau après le *framework* LSM, ils n'ont donc pas été étudiés dans sa conception initiale.

— Contrairement aux interfaces System V, les interfaces POSIX sont développées en s'appuyant sur le système de fichiers. Les files de messages POSIX bénéficient de mécanismes de sécurité par ce biais. En particulier, ouvrir une file de messages POSIX requiert de passer par le crochet LSM d'ouverture des fichiers.

— Les crochets LSM placés dans les appels système `read` et `write` ont essentiellement été placés pour la revalidation des permissions d'accès au fichier pour chaque lecture et écriture. La revalidation est considérée comme pertinente surtout pour les fichiers standards et non les autres types d'objets vivants dans le système de fichiers. Par conséquent, le crochet LSM pour l'ouverture des files de messages est considéré suffisant.

Toutes ces raisons sous-tendent en réalité que l'implémentation du contrôle de flux d'information n'est pas un usage prévu dans le *design* de LSM.

5.6.2 `tee`, `splice` et `vmsplice`

Les appels système `tee`, `splice` et `vmsplice` font conceptuellement des opérations que `read` et `write` pourraient faire. Ces appels tirent parti de l'implémentation des tubes dans le noyau Linux pour faire des copies d'information entre tubes, ou entre un tube et un fichier standard, sans réaliser réellement de copie octet à octet dans la mémoire. Les raisons pour lesquels certains crochets manquent rejoignent celles avancées pour les files de messages POSIX. Les tubes sont considérés comme n'étant pas sujet à revalidation. Par conséquent, lorsqu'un appel système n'est utilisé que pour la lecture ou l'écriture dans des tubes, les crochets LSM ont été omis. On peut par ailleurs constater une limitation de notre analyse statique ici. Nous avons vérifié la présence de crochets LSM dans les appels système mais nous n'avons pas vérifié si les crochets sont adéquats pour observer le flux causé par les appels système. Par exemple, lorsque `splice` est utilisé pour causer un flux d'un tube vers un fichier, un seul crochet est traversé : le crochet normalement utilisé par les modules de sécurité pour vérifier la permission d'écriture dans le fichier. Le moniteur de flux d'information pourrait donc se méprendre et constater un flux du processus appelant vers le fichier, au lieu du tube vers le fichier. Une implémentation de moniteur correcte requiert donc un moyen de distinguer les différents usages d'un même crochet du point du vue du suivi de flux.

5.6.3 `process_vm_readv`, `process_vm_writev`, `ptrace`

Ces trois appels système permettent de faire des flux de mémoire à mémoire entre deux processus. À première vue, certains chemins d'exécutions ne sont pas couverts. Néanmoins, une analyse manuelle nous a conduit à identifier ces chemins comme correspondant à une situation particulière : un processus lisant sa propre mémoire (c'est-à-dire un *thread* lisant la mémoire d'un *thread* à l'intérieur d'un même processus). Ceci ne correspond pas à un flux d'information d'après notre définition car les *threads* partagent de toute façon leur mémoire, qui forme un unique conteneur d'information.

5.7 Conclusion

Le travail présenté dans ce chapitre a été présenté à la conférence FormaliSE 2017 [36]. Nous avons listé les appels système produisant des flux et nous avons étudié tous les chemins d'exécution les générant. Nous pouvons conclure que LSM est globalement bien adapté au suivi de flux d'informations, à quelques exceptions près que notre approche permet de corriger. Sur la base de nos résultats, nous avons produit les *patchs* pour le noyau Linux corrigeant les problèmes que nous avons identifiés et nous avons revalidé le nouveau noyau en répétant notre analyse sur le framework LSM complété. Des travaux sont cependant encore nécessaires pour poursuivre l'analyse. En effet, nous avons vérifié que les chemins d'exécution produisant des flux sont bien couverts par l'ensemble de crochets appropriés. Le cas de `splice` est problématique par exemple : même si le flux est entre un fichier et un tube, le seul crochet LSM présent est `file_permission` qui normalement caractériserait un flux du fichier vers le processus appelant. Ce cas est unique à notre connaissance, et nous pouvons donc le traiter sans peine, mais une sémantique claire des appels système en termes de flux d'information serait nécessaire pour s'en assurer.

L'avantage clair de notre approche est que nous n'avons pas besoin de formaliser une sémantique complète pour le dialecte du langage C (comportant des extensions implémentées uniquement par GCC) utilisé pour le développement du noyau. D'autre part, être intégré à la chaîne de compilation nous permet de bénéficier de beaucoup de structures de données internes comme les graphes de flots de contrôle, l'analyse des boucles ou encore l'oracle d'alias. De plus, comme ces structures sont celles générées par GCC, le code qui est analysé n'est peut-être pas celui qui est écrit, et peut ne pas correspondre à l'intention du programmeur, mais il s'agit du code qui sera exécuté, ou en tout cas d'une représentation plus proche de ce code que de celui d'origine. Bien que des *frameworks* d'analyse comme BLAST [4] soient beaucoup plus complets et puissants, la sémantique qu'ils donnent au langage C peut différer de celle de GCC et ils ne permettent pas de bénéficier des structures de données internes du compilateur. Enfin, l'utilisation d'un greffon GCC nous permet de nous insérer à n'importe quel endroit de la chaîne de compilation. En particulier, s'insérer relativement tard permet d'analyser un code pré-travaillé, plus long mais comportant moins de types d'instructions différentes, et où tous les branchements et boucles sont transformés en instructions de type *goto*. L'inconvénient majeur de l'usage des greffons est que comme la compilation s'effectue fichier par fichier, les possibilités d'analyses statiques trans-unités de traduction sont limitées. De plus, déboguer les analyses revient à déboguer GCC tout entier, ce qui s'avère assez difficile. L'utilisation d'analyses statiques « assistées par le compilateur » nous semble une approche prometteuse pour développer rapidement et de manière sûre des analyses de taille modeste, aux objectifs restreints.

Chapitre 6

Rfblare : une implémentation de Blare à même de gérer la concurrence entre appels système et les projections en mémoire

Dans le chapitre précédent, nous avons identifié une condition nécessaire pour l'implémentation correcte de moniteurs de flux d'information : la présence d'un crochet LSM dans chaque appel système provoquant un flux d'information. Cette condition garantit qu'il est possible pour un moniteur implémenté avec LSM d'observer tous les flux directs. Cependant, cette condition n'est pas *suffisante* pour garantir l'observation de tous les flux. En effet, nos expérimentations sur des implémentations de flux d'information nous ont conduit à identifier des situations où, bien que tous les flux d'information individuels soient observés, des flux indirects échappent à la vigilance du moniteur de flux d'information. Ces cas se sont révélés symptomatiques de *conditions de concurrence* entre les appels systèmes opérant sur les mêmes conteneurs d'information. Pour illustrer ce problème, on peut considérer la situation présentée en figure 6.1.

Le processus *cat* copie un *fichier* dans *tube* (dans la ligne de commande, le tube est anonyme et créé par l'emploi de « | »), que le processus *wc -l* lit ensuite pour calculer le nombre de lignes du texte. Le processus *wc -l* affiche le résultat sur la sortie standard */dev/stdout*. Trois des conteneurs sont marqués initialement avec un symbole, également appelé *teinte* indiquant la classe d'information enregistrée dans le conteneur : *file* avec $\star$, *cat* avec †, *wc -l* avec •. La tâche du moniteur de flux d'information est de propager les marques d'un conteneur à l'autre lors d'un flux pour enregistrer la diffusion de l'information ; cette opération est appelée la *propagation de teintes*. Si tous les flux étaient observés dans l'ordre où ils sont effectués, le moniteur devrait copier la marque $\star$ du *fichier* à *cat*, puis la marque †$\star$ de *cat* à *tube*, puis à *wc -l* et enfin, la marque •†$\star$ de *wc -l* à */dev/stdout*. Cependant, dans les appels système, le point où le flux est observé, c'est-à-dire le crochet LSM, et le point où le flux est réellement effectué (typiquement un appel de fonction plus tard dans l'appel) ne sont pas exécutés de manière atomique.

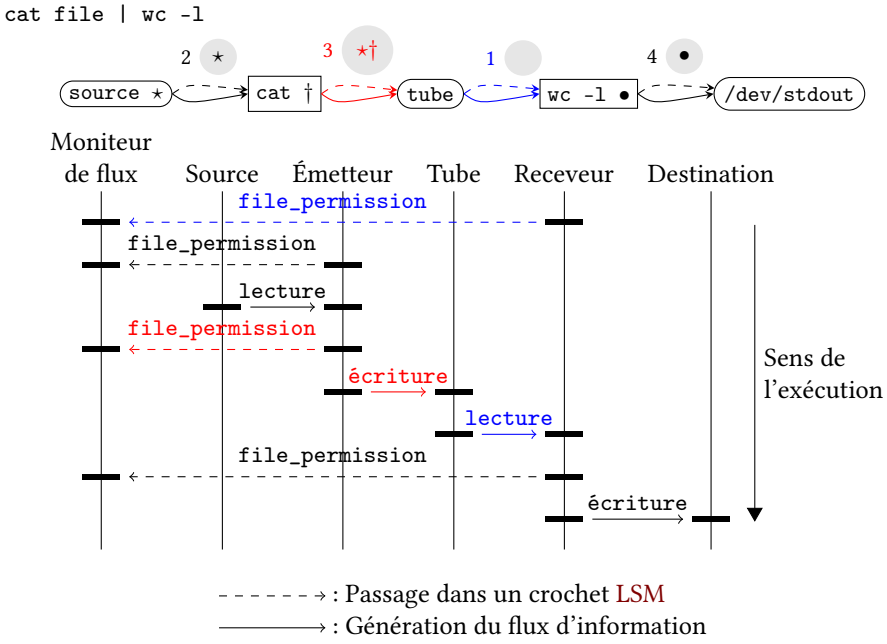


FIGURE 6.1 – Exemple d'exécution problématique

Il est donc tout à fait possible d'observer une séquence d'opérations comme celle détaillée dans le tableau de la figure 6.1. L'exécution des flux et leur observation se font dans un ordre différent, par conséquent, la propagation est incorrecte. Le premier schéma montre la différence entre le résultat de la propagation de teintes et la diffusion de l'information. Les flèches en pointillés représentent la propagation, effectuée dans l'ordre indiquée par les numéros, les flèches pleines indiquent le flux effectués, dans l'ordre de gauche à droite. On constate que bien que l'information soit transférée du fichier *source* vers *destination*, les teintes ne sont pas propagées de manière correspondante. Normalement, `/dev/stdout` devrait recevoir les teintes ⋆ † • ; en réalité, ici, il ne reçoit que • donc le flux indirect de la source et de `cat` vers `/dev/stdout` sont finalement manquants.

Ce problème survient en raison d'une *condition de concurrence* portant sur le conteneur d'information *tube*. Comme la séquence ⟨observation du flux, génération du flux⟩ n'est pas atomique, il est possible d'observer deux flux directs dans un ordre différent de leur génération, et par conséquent, de manquer un flux indirect. Même si un moniteur de flux d'information pouvait tolérer un certain manque de précision, le problème des conditions de concurrence se pose de manière plus sérieuse si l'on considère les flux continus. En effet, si deux processsus partagent une mémoire partagée, alors il est essentiel de considérer que les flux atteignant chacun d'eux atteignent aussi l'autre. Imaginons qu'un des processus lise une donnée depuis un fichier par exemple. Il peut choisir comme tampon de mémoire destination de l'appel système `read` une portion de la mémoire partagée. Il faut également considérer que si deux processus A et B partagent une zone de mémoire d'une part, et que les processus B et C partagent une autre zone d'autre part, alors les processus A et C partagent une zone de mémoire de fait, bien qu'aucun des deux processus ne l'ait créée explicitement.

Nous avons étudié trois moniteurs de flux d'information implémentés avec LSM :

Laminar [76] et *KBlare* [33], développés pour le noyau Linux générique, et *Weir* [67] développé pour le noyau Linux Android. Dans la première sous-section, nous donnons deux exemples d'attaques exploitant la condition de concurrence et les flux continus pour échapper aux moniteurs de flux d'information. Ensuite, nous décrivons un modèle de propagation de teintes pour formaliser :

1. ce que serait une propagation idéale, correspondant exactement aux flux réalisés ;
2. la propagation de teintes réalisée par les moniteurs de flux d'information étudiés ;
3. la différence entre ces deux propagations.

Ce modèle formel nous a permis de développer une solution pour permettre une propagation correcte, à défaut d'être aussi précise que la propagation idéale, c'est-à-dire une propagation calculant une surapproximation des teintes des conteneurs de flux d'information lorsque l'on ne connaît pas l'ordre exact des flux. Cette solution repose sur une intuition simple. Comme il est impossible de faire coïncider l'observation des flux avec leur génération, nous ajoutons des points d'observation de la *fin* des flux. Nous considérons qu'entre les points où le début et la fin du flux sont observés, il est *activé* et peut avoir lieu zéro, une ou plusieurs fois, à n'importe quel instant. Nous calculons la propagation de teintes en tenant compte de tous les flux *activés* simultanément. Cette solution nous permet donc de gérer à la fois les problèmes de concurrence ainsi que les flux continus, en ne considérant plus les flux comme des événements atomiques. Nous avons démontré la correction de notre algorithme à l'aide de l'assistant à la preuve Coq [93]. Tout au long de ce chapitre, nous accompagnons les éléments de définitions et les théorèmes de leur correspondance en Coq. La description formelle de notre algorithme et sa preuve sont disponibles en annexe C ainsi que sur le site du projet Blare à l'adresse `https://blare-ids.org/rfblare/`. Nous montrons l'implémentabilité et l'utilité de notre nouvel algorithme de propagation dans *Rfblare*, une nouvelle version de *KBlare* implémentant uniquement la propagation de teintes, sans la partie de vérification de la légalité des flux. Nous montrons que *Rfblare* induit un surcoût en performance minimal sur des tests concrets.

6.1 Attaques sur des moniteurs de flux d'information implémentés avec LSM

6.1.1 Exploitation d'une condition de concurrence entre `read` et `write`

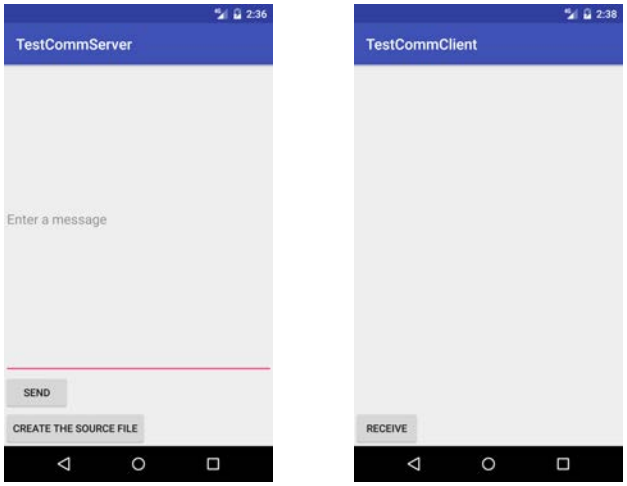
KBlare, *Laminar* et *Weir* utilisent le crochet `file_permission` pour effectuer la propagation de teintes vers ou depuis les fichiers, lors d'une lecture ou une écriture. Deux appels système `read` et `write` peuvent entrer en concurrence s'ils opèrent sur le même fichier. Ceci est vrai y compris sur les systèmes uniprocesseurs. En effet, les appels système peuvent relâcher le CPU et s'endormir entre le passage dans le crochet et l'instruction provoquant le flux. Nous avons testé une attaque similaire à l'exemple présenté dans l'introduction de ce chapitre :

```
mkfifo tube; cat < tube > destination& cat < source > tube
```

Nous créons un tube nommé *tube* permanent dans le dossier où se déroule le test, afin de pouvoir surveiller l'évolution de sa teinte. Avec cette commande, on observe que *KBlare* est incapable de propager les teintes correctement car il observe — dans la plupart des exécutions, le scénario n'étant pas déterministe — la lecture du tube, bloquante, avant son écriture. En réalité, nous avons découvert ce problème de conditions de

concurrence en constatant, au cours d'un test unitaire, que *KBlare* ne transférait pas toujours les teintes correctement lors des lectures–écritures de conteneurs bloquants, comme les tubes et les sockets réseau.

Pour confirmer que le problème n'est pas lié à *KBlare* mais aux moniteurs implémentés avec LSM en général, nous avons répliqué l'attaque sur *Weir*. Ce type d'attaques sur Android est plus délicat à réussir car les processus sont isolés assez strictement par défaut. Nous avons employé un contexte d'attaque un peu artificiel. Nous avons développé deux applications, un client et un serveur (correspondant respectivement au lecteur et à l'écrivain du tube), présentées dans la figure 6.2. Les deux applications sont installées avec le même identifiant utilisateur, de sorte qu'elles partagent un même



(a) Lecteur du tube (b) Écrivain du tube

FIGURE 6.2 – Applications développées pour mener l'attaque sur *Weir*

dossier d'installation. Pour jouer l'attaque, il faut, manuellement, créer un tube dans ce dossier. Au démarrage, l'écrivain, l'application *TestCommServer* demande une teinte au système *Weir*. Par la suite, l'utilisateur peut écrire un message dans une zone de texte puis appuyer sur le bouton « Create the source file ». Ce bouton crée un fichier source, teinté selon la valeur retournée par *Weir* et contenant le texte entré dans la boîte de dialogue. Ensuite, un appui sur le bouton « Send » copie le fichier dans le tube. Le lecteur du tube, l'application *TestCommClient*, possède une zone de texte et un bouton « Receive ». L'appui sur ce bouton déclenche la lecture du tube et l'affichage de son contenu dans la zone de texte, confirmant le flux d'information indirect de l'écrivain au client via le tube. La propagation de teintes peut être observée dans l'interface de débogage, en observant la trace fournie par *Weir* et retranscrite avec des commentaires dans l'extrait 6.1. On observe un phéomène intéressant : si l'on clique sur le bouton de réception en premier, la teinte de l'écrivain n'est pas propagée jusqu'au lecteur ; en revanche, si l'on clique sur le bouton d'émission avant le bouton de réception, la propagation de teintes est conforme aux flux réalisés. Cela confirme expérimentalement l'analyse de la figure 6.1. La condition de concurrence est ici particulièrement facile à déclencher car les lectures depuis les tubes vides sont bloquantes jusqu'à ce qu'un autre processus écrive dedans. Nous avons exploité ce fait pour faciliter les attaques

```
1  === Création du fichier source par l'émetteur ===
2  <4>[ 3060.824300] WEIR_DEBUG: File Permission. pid 11372,
       file /data/data/com.example.lgeorget.testcommclient/files
       /source
   <4>[ 3060.824525] WEIR_DEBUG: Label on pid 11372 is:
       {96263937442022980, }

5  === Lecture par le receveur du tube ===
   <4>[ 3065.723624] WEIR_DEBUG: File Permission. pid 11462,
       file /data/data/com.example.lgeorget.testcommclient/files
       /tube
   === pas de label sur le tube ni sur le receveur ici ===

   === Lecture par l'émetteur de source ===
10 <4>[ 3068.259636] WEIR_DEBUG: File Permission. pid 11372,
       file /data/data/com.example.lgeorget.testcommclient/files
       /source
   <4>[ 3068.260424] proc_label={96263937442022980, }
   <4>[ 3068.261075] file_label={96263937442022980, }

   === Écriture par l'émetteur dans le tube ===
15 <4>[ 3068.261268] WEIR_DEBUG: File Permission. pid 11372,
       file /data/data/com.example.lgeorget.testcommclient/files
       /tube
   <4>[ 3068.261486] WEIR_DEBUG: Label on pid 11372 is:
   <4>[ 3068.261524] {96263937442022980, }

   === Écriture par le receveur de destination ===
20 <4>[ 3068.266815] WEIR_DEBUG: File Permission. pid 11462,
       file /data/data/com.example.lgeorget.testcommclient/files
       /destination
   <4>[ 3068.267042] WEIR_DEBUG: Label on pid 11462 is: {}
   === le receveur doit transmettre le tag à la destination ici
       mais il ne l'a pas acquise ===
```

Extrait 6.1 – Extrait des traces émises par *Weir* durant l'exécution réussie de l'attaque utilisant *TestCommClient* et *TestCommServer*

mais il est important de noter qu'avec un fichier standard, la condition de concurrence existerait également, pour *KBlare* comme pour *Weir*.

Dans le cas de *Laminar*, cette attaque n'est pas directement possible car les développeurs ont ajouté une synchronisation aux appels système *read* et *write*, de sorte que l'exécution de chaque opération de lecture ou d'écriture soit atomique. Cette solution prévient effectivement toute possibilité de condition de concurrence mais nous ne considérons pas néanmoins qu'elle soit satisfaisante dans tous les cas. D'une part, elle diminue les performances en sacrifiant le parallélisme d'appels très fréquents. Une lecture depuis un système de fichiers avec une certaine latence, comme un système de fichiers sur le réseau par exemple, bloquerait toutes les autres lectures–écritures. D'autre part, cela altère la sémantique de certains conteneurs d'information qui sont censés bloquer la lecture quand ils sont vides, comme les tubes et les sockets réseaux. Cette sémantique est attendue et exploitée par de nombreuses applications. Ce n'est pas un problème pour *Laminar* qui impose de toute manière l'usage des tubes de manière non-bloquantes pour éviter un canal caché de flux d'information, mais cela est rédhibitoire pour *KBlare* et *Weir* qui ne veulent pas requérir de portage des applications. De plus, si l'on étend cette solution à tous les appels système pouvant entrer en concurrence, le risque de causer des interblocages devient grand.

6.1.2 Exploitation de flux d'information continus

Lire ou écrire un fichier peut être fait avec les appels système de la famille de `read` et `write` mais ce n'est pas la seule méthode. Il est possible pour un processus de projeter un fichier dans sa mémoire, c'est-à-dire de demander au noyau, via l'appel système `mmap`, de faire correspondre une certaine portion de son espace d'adressage à une plage de même longueur dans le fichier, de sorte que les accès à ces adresses soient traduits par le noyau en des accès au fichier sous-jacent. Cette opération est très courante ; en fait, c'est de cette manière que les processus chargent leur exécutable en mémoire. Ce mécanisme permet également la mise en place de mémoires partagées. Il existe deux interfaces de manipulation des mémoires partagées offertes aux processus : l'ancienne interface System V et la nouvelle interface POSIX. Elles partagent leur implémentation interne mais s'utilisent différemment. Les mémoires partagées System V ont une collection d'appels système propres et possèdent chacune un identifiant unique tandis que les mémoires POSIX ne sont qu'une abstraction autour des projections de fichiers en mémoire. Il n'y pas d'appels système dédiés pour les manipuler, uniquement des fonctions de la bibliothèque standard. En interne, elles réalisent la projection avec `mmap` d'un certain fichier commun aux processus désireux de partager de la mémoire.

Les flux continus ne sont pas pris en compte par *Weir* qui se concentre essentiellement sur les fichiers et les processus. *Laminar* non plus ne propose pas de solutions à ce problème. En revanche, on peut remarquer que le problème est connu des développeurs. Dans le code source de *Laminar*, on trouve le commentaire suivant : « XXX : Should do something about mmaped files. » [75, fichier `security/difc.c`, l. 944]. Cela laisse penser que le problème est considéré comme important par les développeurs mais n'est pas trivial à résoudre. *KBlare* propose quelques solutions pour propager les teintes à travers les projections et mémoires partagées. En premier lieu, lorsqu'un processus lit un fichier, les teintes sont propagées non seulement au processus mais aussi aux fichiers qu'il projette avec les droits d'écriture. Il est également décrit dans la thèse de Hauser que *KBlare* gère les mémoires partagées de type System V en maintenant à jour la liste des identifiants de mémoires partagées associées à chaque processus et en utilisant cette liste pour propager les teintes lorsqu'un flux atteint un processus [38].

L'implémentation est cependant incomplète sur ce point et ne met pas en œuvre l'intégralité du mécanisme [39]. D'autre part, sa conception est en partie problématique aussi. *KBlare* échoue à prendre en compte la « transitivité » entre mémoires partagées : deux processus peuvent être liés par une chaîne de mémoires partagées et de processus et donc communiquer de fait via une zone de mémoire partagée « composée » qu'aucun des deux n'a mis en place.

Pour confirmer l'utilisabilité des flux continus pour échapper à la vigilance des moniteurs de flux d'information, nous avons mis au point une variante de l'attaque précédente utilisant les projections en mémoire de fichiers ainsi que des mémoires partagées. La situation est présentée dans la figure 6.3. Nous remplaçons le fichier *source* par une projection en lecture seule dans la mémoire du processus émetteur, le fichier *destination* par une projection en lecture–écriture dans la mémoire du processus récepteur et enfin le tube par une zone de mémoire partagée en lecture–écriture par l'émetteur et le récepteur.

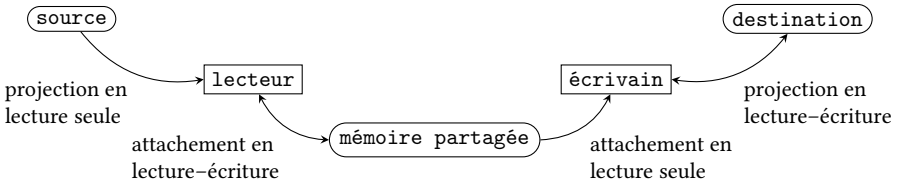


FIGURE 6.3 – Description de la mise en place de l'attaque via les projections de fichiers et mémoires partagées

Nous n'avons pas pu tester directement l'attaque sur les moniteurs de flux présentés en raison du manque d'implémentation fonctionnelle. Nous avons réalisé une implémentation minimale de la propagation de *KBlare* décrite dans le travail de HAUSER [38] pour jouer l'attaque. Comme prévu, quel que soit l'ordre dans lequel sont réalisées les projections et la mise en place de la mémoire partagée, le flux du fichier *source* au fichier *destination* est possible. En revanche, la propagation de teinte ne se fait pas correctement si la projection du fichier *source* dans le processus lecteur est faite en dernier. Dans ce cas précis, lors de la projection, la teinte est correctement propagée jusqu'au processus lecteur et jusqu'à la mémoire partagée qui y est attachée, mais n'atteint pas le fichier *destination* projeté dans le processus écrivain attaché à la mémoire partagée. Avec une chaîne de projections suffisamment longue, il est donc possible de tromper le mécanisme.

6.2 Algorithme de propagation de teinte

Nous proposons un nouveau modèle formel des flux d'information entre conteneurs ainsi qu'une description formelle des mécanismes de propagation de teintes en ne considérant pas que les flux sont des événements atomiques de la vie du système mais au contraire des opérations successivement *activées*, *exécutées* et *désactivées*.

6.2.1 Tags, flux et exécutions

On note $\mathcal{C}$ l'ensemble des conteneurs d'information d'un système, passés, présents ou à venir. Dans notre modèle, tous les conteneurs existent en permanence. Il suffit de

considérer les conteneurs n'existant plus ou pas encore comme des conteneurs ne pouvant être la source ou destination d'aucun flux. Les conteneurs d'information stockent ou acheminent de l'information, provenant de l'utilisateur, d'autres machines, etc. Les fichiers, l'espace mémoire des processus, les files de messages sont des conteneurs.

Le moniteur de flux d'information encode les flux dans le système et l'impact qu'ils ont sur les conteneurs en attachant à chacun d'eux une teinte. Dans notre modèle générique, nous considérons qu'une teinte est un ensemble de *tags*. Nous considérons, sans perdre en généralité, que tout conteneur $c \in \mathcal{C}$ est initialement porteur d'un unique tag noté t^c. Ce tag identifie les informations originaires de c. On note $\mathcal{T} = \uplus_{c \in \mathcal{C}} \{t^c\}$ l'ensemble des tags ($\uplus$ dénotant l'union disjointe). Durant la vie du système, des conteneurs sont créés et détruits et des flux ont lieu entre eux. La propagation de teinte consiste à modifier l'ensemble de tags du processus destination d'un flux afin d'enregistrer le fait qu'il pourrait dorénavant contenir des informations originaires de la source.

Définition 17 (Configuration, Teinte)**.** Une configuration $\theta : \mathcal{C} \to \wp(\mathcal{T})$ associe à chaque conteneur un ensemble de tags. $\theta(c) = \{t^{c_1}, \ldots, t^{c_n}\}$ est la teinte de c et indique intuitivement que c a été la destination de flux originaires de c_1, …, c_n.

On note l'ensemble de toutes les configurations possibles Θ. On distingue une configuration particulière : θ_{init}, qui est la configuration initiale du système telle que $\forall c \in \mathcal{C}\ \theta_{init}(c) = \{t^c\}$. Intuitivement, une configuration représente une abstraction de l'état du système. Cet état évolue lorsqu'un événement de flux a lieu.

En Coq, ces ensembles sont définis comme suit :

```
Variable Tag : Set.
Variable Container : Set.
Variable sourceTag : Container → Tag.
Inductive initConfiguration : Configuration :=
| init c :
 initConfiguration c (sourceTag c).
```

Nous considérons trois types d'événements relatifs aux flux : l'activation, l'exécution et la désactivation. Naturellement, un flux ne peut être exécuté ou désactivé qu'une fois activé, et ne peut plus être exécuté une fois désactivé. De plus, chaque flux est unique, on ne peut pas réactiver un flux désactivé. Comme plusieurs flux entre deux mêmes conteneurs peuvent avoir lieu au cours de la vie du système, on identifie chaque flux de manière unique par un élément de l'ensemble $\mathcal{F}$ des identifiants de flux.

Définition 18 (Événements)**.** Soient $c_1, c_2 \in \mathcal{C}$, et $f \in \mathcal{F}$. On définit une relation $c_1 \to_f c_2$ décrivant un flux appelé f de c_1 à c_2. Un événement $e \in \mathcal{E}$ est soit une paire $(f, (c_1, c_2))$ avec $c_1 \xrightarrow{enable}_f c_2$ ou $c_1 \xrightarrow{disable}_f c_2$, soit une paire $(f, (c_1, c_2))$ avec $c_1 \xrightarrow{exec}_f c_2$. Le premier ensemble est appelé $\mathcal{O}$ et le second $\mathcal{X}$. Ces relations ont intuitivement le sens suivant :

$c_1 \xrightarrow{enable}_f c_2$ signifie que le flux identifié par f de source c_1 et destination c_2 est activé ;

$c_1 \xrightarrow{exec}_f c_2$ signifie que le flux identifié par f de source c_1 et destination c_2 est exécuté ;

$c_1 \xrightarrow{disable}_f c_2$ signifie que le flux identifié par f de source c_1 et destination c_2 est désactivé.

$\mathcal{O}$ contient les événements d'activation et de désactivation des flux tandis que $\mathcal{X}$ contient les événements d'exécution.

L'évolution du système causée par les flux est représentée par une *exécution*. On note $\mathcal{E}^+$ l'ensemble des séquences d'événements non vides de $\mathcal{E}$ et nous considérons $\mathbf{E} \subset \mathcal{E}^+$ un sous-ensemble strict de ces séquences, que nous appelons *exécutions*. On adopte les notations suivantes :

- $e[i]$ est le i-ème événement de l'exécution $e \in \mathbf{E}$;
- $\lg(e)$ est la longueur de e, c'est-à-dire le nombre d'événements qui la composent ;
- $e[:n]$ est le préfixe $(e[1], \ldots, e[n])$ de longueur n de e.

En Coq, les identifiants de flux constituent un ensemble sans hypothèse particulière. Les événements sont définis comme un type algébrique et les exécutions sont des listes (des séquences) d'événements. Pour faciliter les définitions, on permet les exécutions vides dans la définition.

```
Variable Flow : Set.

Inductive Event : Set :=
| enable : Container → Flow → Container → Event
| disable : Container → Flow → Container → Event
| exec : Container → Flow → Container → Event

Definition Execution : Set := list Event.
```

Le sous-ensemble $\mathbf{E}$ de toutes les séquences d'événements possibles est caractérisé par deux conditions de causalité, en cohérence avec la définition des événements donnée plus haut. L'activation d'un flux précède son exécution, qui précède sa désactivation, dans toute exécution.

$$\forall i \;\; e[i] = c_1 \xrightarrow{disable}_f c_2 \vee e[i] = c_1 \xrightarrow{exec}_f c_2 \Rightarrow \left(\exists j < i \;\; e[j] = c_1 \xrightarrow{enable}_f c_2 \wedge \left(\forall k \; j < k < i \Rightarrow (e[k] \neq c_1 \xrightarrow{disable}_f c_2)\right)\right) \tag{6.1}$$

En Coq, la propriété de causalité est exprimée de manière inductive. L'exécution vide est causale, ainsi que toute exécution composée d'une sous-exécution causale à laquelle on rajoute un événement respectant quelques hypothèses (on peut ajouter la désactivation que si l'activation est dans la sous-exécution, par exemple).

```
Inductive causality : Execution → Prop :=
| causality_nil : causality []
| causality_enable c1 f c2 e
    (Hind: causality e)
    (Hnoenable: ¬ In (enable c1 f c2) e) :
    causality (enable c1 f c2 :: e)
| causality_exec c1 f c2 e
    (Hind: causality e)
    (Henable: In (enable c1 f c2) e)
    (Hnodisable: ¬ In (disable c1 f c2) e) :
    causality (exec c1 f c2 :: e)
| causality_disable c1 f c2 e
    (Hind: causality e)
```

```
    (Henable: In (enable c1 f c2) e)
    (Hnodisable: ¬ In (disable c1 f c2) e) :
    causality (disable c1 f c2 :: e).
```

Nous supposons que tous les événements d'une exécution ne sont pas observables par un moniteur de flux d'information. Par exemple, un moniteur utilisant LSM ne peut consulter et modifier l'état du système que lors du passage dans un crochet. En particulier, nous posons que $\mathcal{O}$ est l'ensemble des événements observables tandis que les événements de $\mathcal{X}$ sont cachés, c'est-à-dire qu'ils ne sont pas détectables par le moniteur de flux. Ceci modélise le fait que le moniteur est implémenté avec LSM et qu'il ne peut donc pas observer l'intégralité de l'activité du système. En réalité, il ne peut prendre connaissance de l'état du système et effectuer la propagation de teintes que lorsque l'exécution d'un appel système atteint un crochet LSM. La véritable opération de l'appel système, le flux, est donc pratiquée entre le premier crochet et un second événement permettant au moniteur d'agir, comme un crochet placé à la fin de l'appel système, pour les flux discrets, ou dans l'appel système de fin de flux, pour les flux continus.

On définit naturellement l'observabilité des événements et des événements en Coq. La fonction `flatten_list` sert à satisfaire le système de type de Coq car `causality` s'applique sur des listes d'événements et non des listes d'événements observables.

```
Inductive observable : Event → Prop :=
| enable_is_obs c1 f c2 :
  observable (enable c1 f c2)
| disable_is_obs c1 f c2 :
  observable (disable c1 f c2).
Inductive hidden : Event → Prop :=
| exec_is_hidden c1 f c2 :
  hidden (exec c1 f c2).

Definition ObsExecution :=
 { e:list { ev:Event | observable ev } |
   causality (flatten_list e) }.
Definition HidExecution := list { ev:Event | hidden ev }.
```

On note $\mathbf{O}$ l'ensemble des exécutions observables, c'est-à-dire ne contenant que des événements dans $\mathcal{O}$, et $\mathbf{X}$ l'ensemble des exécutions cachées, ne contenant que des événements de $\mathcal{X}$. Les exécutions observables représentent ce qu'un moniteur de flux d'information est capable de voir tandis que les exécutions cachées sont celles qui représentent l'évolution effective des conteneurs d'information. Soit $e \in \mathbf{E}$, on écrit $e_{\mathcal{O}} \in \mathbf{O}$ (respectivement $e_{\mathcal{X}} \in \mathbf{X}$) l'exécution observable (respectivement l'exécution cachée) obtenue en retirant les événements cachés (respectivement les événements observables) de e. Ces deux projections de l'exécution e sont liées par les conditions de causalité exposées dans l'équation (6.1). On définit par conséquent une relation de compatibilité entre une exécution observable et une exécution cachée.

Définition 19 (Compatibilité)**.** Une exécution observable ω est compatible avec une exécution cachée x si, et seulement si, elles sont la projection d'une même exécution de $\mathbf{E}$. Formellement,
$\forall x \in \mathbf{X}\ \forall \omega \in \mathbf{O}$, on écrit $\omega \vdash x$ ssi $\exists e \in \mathbf{E}\ (\omega = e_{\mathcal{O}} \wedge x = e_{\mathcal{X}})$.

En Coq, nous définissons la compatibilité de manière inductive. En quelque sorte, la définition donne une relation entre une exécution et les deux sous-exécutions observable et cachée qui la composent, en fonction de l'exécution avec un élément de moins. Nous prouvons ensuite un théorème montrant que cette définition est équivalente à la définition 19 : si on a relation de compatibilité entre une exécution observable, une exécution cachée et une exécution, alors cette exécution respecte la causalité et elle est composée des événements des exécutions cachée et observable.

```
Inductive compatible : ObsExecution → HidExecution → Execution → Prop :=
| compatible_empty :
  compatible (exist _ [] causality_nil) [] []
| compatible_add_exec
  o x e c1 f c2
  (Hcompatible: compatible o x e)
  (Henabled: In (make_enable c1 f c2) (‘ o))
  (Hnot_disabled: ¬ In (make_disable c1 f c2) (‘ o)) :
  compatible o (make_exec c1 f c2 :: x) (exec c1 f c2 :: e)
| compatible_add_enable
  o x e c1 f c2
  (Hcompatible: compatible o x e)
  (Hno_enabled: ¬ In (make_enable c1 f c2) (‘ o)) :
  compatible (make_enable_causality c1 f c2 o Hno_enabled) x (enable c1 f c2 :: e)
| compatible_add_disable
  o x e c1 f c2
  (Hcompatible: compatible o x e)
  (Henabled: In (make_enable c1 f c2) (‘ o))
  (Hnot_disabled: ¬ In (make_disable c1 f c2) (‘ o)) :
  compatible (make_disable_causality c1 f c2 o Henabled Hnot_disabled) x (
    disable c1 f c2 :: e).

Program Theorem compatibility_implies_interleaving_respecting_causality :
∀(o:ObsExecution) (x:HidExecution) (e:Execution), compatible o x e → (causality e
    ∧
∀ev, (In ev (flatten_list (‘ o)) ∨ In ev (flatten_list x)) ↔ In ev e).
```

Exemple 1. On considère la première attaque présentée dans la section 6.1.1 et illustrée dans la figure 6.4. Dans le reste de cet article, on abrège le nom des conteneurs de la manière suivante : src est la *source*, se le processus *lecteur* de la *source* (et écrivain du tube), p le *tube*, r le processus *écrivain* de *destination* (et lecteur du tube) et d la *destination*. La colonne x représente l'exécution cachée tandis que la colonne ω correspond à l'exécution observable. Ces deux exécutions sont compatibles car l'exécution de la colonne e entrelace x et ω de telle sorte que les conditions de causalité de (6.1) soient respectées.

6.2.2 Interprétation des exécutions en termes de flux d'information

On dit d'un mécanisme de propagation de teintes qu'il est *correct* s'il prend en compte tous les flux ayant eu lieu, c'est-à-dire s'il ne permet pas qu'il existe une exécution cachée compatible avec l'observation observée par le moniteur, telle qu'un

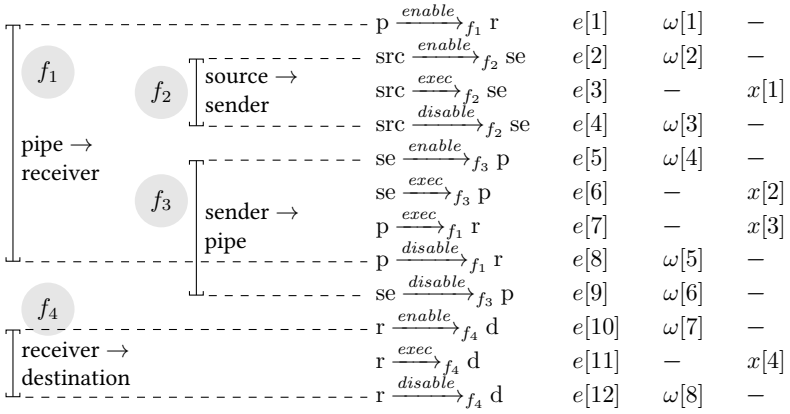


FIGURE 6.4 – Exécutions observable et cachée de l'exemple d'attaque

flux exécuté n'est pas reflété dans la propagation de teinte. Naturellement, une exécution observable peut être compatible avec plusieurs exécutions cachées, dès lors que plusieurs flux sont activés simultanément. Une exécution cachée est en fait un ordre total sur les flux se produisant dans le système tandis qu'une exécution observable est un ordre partiel sur ces flux : certains flux apparaissent en concurrence aux yeux du moniteur de flux, ils ne peuvent pas être ordonnés temporellement. L'ensemble des exécutions cachées compatibles avec une exécution observable est exactement l'ensemble des ordres totaux compatibles avec cet ordre partiel. Par conséquent, un moniteur de flux d'information ne peut pas prétendre à une propagation de teinte exacte dans le cas général. La propriété de correction garantit que la teinte calculée pour chaque conteneur est un sur-ensemble de la teinte qui correspondrait à l'exécution cachée.

6.2.3 Propagation idéale

On définit une relation de transition $\hookrightarrow \subseteq \Theta \times \mathcal{X} \times \Theta$ entre configurations décrivant comment un moniteur de flux d'information qui pourrait observer les exécutions des flux eux-mêmes plutôt que leur activation et désactivation effectuerait la propagation de teinte.

$$\theta \xhookrightarrow{c_1 \xrightarrow{exec}_f c_2} \theta[c_2 \leftarrow \theta(c_2) \cup \theta(c_1)]$$

La propagation de teintes consiste simplement à ajouter au label de la destination tous les tags de la source. Soit $x \in \mathbf{X}$, on note $\theta_0 \xhookrightarrow{x[:n]} \theta_n$ lorsque l'on a $\theta_0 \xhookrightarrow{x[1]} \theta_1 \xhookrightarrow{x[2]} \cdots \theta_{n-1} \xhookrightarrow{x[n]} \theta_n$. En reprenant l'exemple 1 page 137, le tableau 6.1a détaille le calcul de propagation idéale effectuée par $\xhookrightarrow{x}$.

En Coq, nous définissons une exécution cachée comme une relation entre configurations. Deux configurations sont en relation, si la configuration d'arrivée est identique à la configuration de départ exceptée pour ce qui est de la destination du flux, qui comprend tous les tags de la source de celui-ci.

```
1 Inductive ConcreteExecution1 : Configuration → { e : Event | hidden e } →
    Configuration → Prop :=
```

TABLE 6.1 – Interprétations des exécutions.
Dans les tableaux suivants, on note : $t^{src} = ★$, $t^{se} = ■$, $t^{p} = □$, $t^{r} = ◎$, $t^{d} = ▲$

(a) Calcul de $\theta_{init} \xrightarrow{x[:n]} \theta$ (Propagation idéale)

n	$x[n]$	$\theta(src)$	$\theta(se)$	$\theta(p)$	$\theta(r)$	$\theta(d)$
1	$src \xrightarrow{exec}_{f_1} se$	★	■, ★	□	◎	▲
2	$se \xrightarrow{exec}_{f_2} p$	★	■, ★	□, ■, ★	◎	▲
3	$p \xrightarrow{exec}_{f_3} r$	★	■, ★	□, ■, ★	◎, □, ■, ★	▲
4	$r \xrightarrow{exec}_{f_4} d$	★	■, ★	□, ■, ★	◎, □, ■, ★	▲, ◎, □, ■, ★

(b) Calcul de $\theta_{init} \xrightarrow{\omega[:n]} \theta$ (Moniteurs de flux reposant sur LSM)

n	$\omega[n]$	$\theta(src)$	$\theta(se)$	$\theta(p)$	$\theta(r)$	$\theta(d)$
1	$p \xrightarrow{enable}_{f_1} r$	★	■	□	◎, □	▲
2	$src \xrightarrow{enable}_{f_2} se$	★	■, ★	□	◎, □	▲
4	$se \xrightarrow{enable}_{f_3} p$	★	■, ★	□, ■, ★	◎, □	▲
7	$r \xrightarrow{enable}_{f_4} d$	★	■, ★	□, ■, ★	◎, □	▲, ◎, □

```
| exec_conc (theta1:Configuration) c1 f c2 (theta2:Configuration)
 (Htheta:∀c t, theta2 c t ↔ ((c = c2) ∧ theta1 c1 t) ∨ theta1 c t) :
 ConcreteExecution1 theta1 (make_exec c1 f c2) theta2.

Inductive ConcreteExecution : Configuration → HidExecution → Configuration
     → Prop :=
| exec_conc_refl (theta:Configuration) :
 ConcreteExecution theta [] theta
| exec_conc_step (theta1 theta theta2:Configuration) (x:HidExecution)
   (ev: Event) (Hhid: hidden ev)
   (Hstep: ConcreteExecution1 theta (exist _ ev Hhid) theta2)
   (Hind: ConcreteExecution theta1 x theta) :
 ConcreteExecution theta1 (make_hidden_exec x ev Hhid) theta2.
```

6.2.4 Propagation de teintes effectuée par les moniteurs implémentés avec LSM

Nous décrivons à présent la façon dont les moniteurs de flux d'information conçus à base de crochets LSM, comme *Laminar*, *KBlare* ou *Weir* propagent les teintes. La caractéristique principale de ces moniteurs est qu'ils ne reposent que sur des crochets LSM placés avant le flux. Par conséquent, dans notre modèle, cela revient à dire qu'ils ne tiennent compte que des événements d'activation, et qu'ils considèrent ceux-ci comme équivalents aux événements d'exécution des flux. Formellement, le calcul de propagation de teinte réalisé par ces moniteurs peut être décrit par une relation

$\rightharpoonup \subseteq \Theta \times \mathcal{O} \times \Theta$ définie par :

$$\theta \xrightarrow{c_1 \xrightarrow{enable}_f c_2} \theta[c_2 \leftarrow \theta(c_2) \cup \theta(c_1)] \qquad \theta \xrightarrow{c_1 \xrightarrow{disable}_f c_2} \theta$$

Lorsqu'un événement d'activation survient, le moniteur de flux s'empresse d'effectuer la propagation de teintes. En revanche, il ignore les événements de désactivation. En considérant à nouveau la situation de l'exemple 1, le tableau 6.1b décrit le calcul réalisé par $\xrightarrow{\omega}$. Cette propagation n'a pas la propriété de correction : alors que $\omega \vdash x$, le flux indirect de *source* vers *destination* est manqué par $\xrightarrow{\omega}$ ($t^{\mathrm{src}} = \bigstar \in \theta(\mathrm{d}) = \{t^{\mathrm{d}} = \blacktriangle, t^{\mathrm{r}} = \circledcirc, t^{\mathrm{p}} = \square, t^{\mathrm{se}} = \blacksquare, t^{\mathrm{src}} = \bigstar\}$ dans le calcul de $\xrightarrow{x}$, mais ce n'est pas le cas pour $\xrightarrow{\omega}$).

Il existe en réalité deux modèles de suivi des flux au niveau du système d'exploitation : celui des *teintes flottantes* utilisé par *KBlare* et *Weir* et celui des *élévations explicites* utilisé par *Laminar*. Dans le modèle de labels flottants, le premier à avoir été formalisé et utilisé, par exemple dans *IX* [56], dès qu'un flux est détecté, la teinte de la destination est automatiquement augmentée avec la teinte de la source. En revanche, dans un modèle d'élévation explicite, la processus causant le flux doit modifier explicitement la teinte de la destination avant d'effectuer le flux. Ce modèle a été formalisé par Zeldovich et ses collaborateurs, concepteurs d'*HiStar* [109]. Le modèle que nous proposons ici décrit directement le modèle de suivi de flux à teinte flottante mais, de manière moins intuitive, il décrit également correctement le modèle d'élévation explicite de labels. En effet, dans les deux cas, la condition de concurrence que nous illustrons dans la section 6.1.1 existe et a les mêmes effets. Dans *KBlare* et *Weir*, le flux du tube au processus récepteur a lieu sans que la teinte du récepteur ne soit mise à jour correctement car les flux sont observés dans l'ordre inverse de leur exécution. Si ce flux est illégal selon une certaine politique de sécurité en vigueur dans le système, alors la violation de cette politique n'est pas détectée et aucune alerte n'est levée. Dans *Laminar*, même si le flux du tube vers le processus récepteur est illégal *au moment où il est effectué*, ce qui signifie que la teinte du récepteur ne permet pas le flux normalement, il ne l'est pas encore *au moment où il est observé*, et il peut donc être déclenché sans élever la teinte du récepteur et sans lever d'alerte. En fait, dans notre modèle, les teintes représentent la connaissance qu'a le moniteur de flux d'information à propos des flux passés du système, et ne tient pas compte de leur sémantique précise en termes de politiques de flux d'information.

6.2.5 Plus petite surapproximation correcte des teintes à propager

Afin de définir une propagation de teinte correcte, nous définissons en premier lieu l'ensemble de tous les flux, directs et indirects, pouvant être causés par une exécution observable. Nous rappelons que l'ensemble de ces flux correspondent à l'ensemble des flux directs et indirects causés par toutes les exécutions cachées compatibles avec cette exécution observable.

Étant donnée une exécution observable ω, on définit $Enabled_\omega \subseteq \mathcal{C} \times \mathcal{C}$ comme l'ensemble des flux ayant été activés au cours de l'exécution ω mais pas encore désactivés à la fin de celle-ci ω.

$$(c_1, c_2) \in Enabled_\omega \Leftrightarrow \exists i\ \omega[i] = c_1 \xrightarrow{enable}_f c_2 \wedge \forall j > i\ \omega[j] \neq c_1 \xrightarrow{disable}_f c_2$$

$Enabled^*_\omega$ est le fermeture réflexive et transitive de la relation $Enabled_\omega$. L'ensemble des flux pouvant être causés par ω s'écrit $Flows_\omega \subseteq \mathcal{C} \times \mathcal{C}$ et est calculé de la manière suivante. [1]

$$Flows_\omega = \begin{cases} Enabled^*_\omega & \text{si } \lg(\omega) = 1 \\ Flows_{\omega[:k]} \,.\, Enabled^*_\omega & \text{si } \lg(\omega) = k+1 \end{cases}$$

En Coq, nous définissons les ensembles de flux activés comme ci-dessus et l'ensemble des flux passés de manière inductive, en commençant avec l'exécution comportant un seul événement.

```
Inductive Enabled : ObsExecution → Container → Container → Prop :=
| enabled_if_not_disabled_yet o c1 f c2
  (Henabled: In (make_enable c1 f c2) (‘o))
  (Hnot_disabled: ¬ In (make_disable c1 f c2) (‘o)) :
  Enabled o c1 c2.

Inductive Flows : ObsExecution → Container → Container → Prop :=
|  flows_1 (c1' c2' c1 c2:Container) (f:Flow)
    (Henabled: (clos_refl_trans _ (Enabled (make_singleton_causality c1' f c2')))
      c1 c2) :
    Flows (make_singleton_causality c1' f c2') c1 c2
| flows_ind_enabled (o:ObsExecution) (c0 c1 c2 c1' c2':Container) (f:Flow)
    (Hind: Flows o c0 c1)
    (Hnot_enabled: ¬ In (make_enable c1' f c2') (‘ o))
    (Hlast: (clos_refl_trans _ (Enabled (make_enable_causality c1' f c2' o
      Hnot_enabled)) c1 c2)) :
    Flows (make_enable_causality c1' f c2' o Hnot_enabled) c0 c2
| flows_ind_disabled (o:ObsExecution) (c0 c1 c2 c1' c2':Container) (f:Flow)
    (Hind: Flows o c0 c1)
    (Henabled: In (make_enable c1' f c2') (‘ o))
    (Hnot_disabled: ¬ In (make_disable c1' f c2') (‘ o))
    (Hlast: (clos_refl_trans _ (Enabled (make_disable_causality c1' f c2' o
      Henabled Hnot_disabled)) c1 c2)) :
    Flows (make_disable_causality c1' f c2' o Henabled Hnot_disabled) c0 c2.
```

Par exemple, si le flux (A, B) est survenu au cours de l'exécution observable courante, et que le flux (B, C) devient activé, alors la composition (A, C) est un nouveau flux (de même que (B, C) naturellement). Ce ne serait pas le cas si (B, C) était antérieur à (A, B). Toujours en considérant l'exemple 1, le tableau 6.2a illustre le calcul de $Flows_\omega$. $Flows_\omega$ n'est pas nécessairement une relation transitive, car l'ordre des flux compte.

Nous prouvons dans la proposition 6 ci-dessous que ce calcul de $Flows$ garantit la correction de la propagation de teintes associée, illustrée dans le tableau 6.2b. La proposition 7 garantit quant à elle qu'il est impossible de calculer une surappoximation plus petite des teintes garantissant la correction dans notre modèle. En d'autres termes, pour tout flux apparaissant dans $Flows_\omega$ il existe une exécution cachée compatible avec ω causant ce flux. Dans la mesure où ces propositions ont été démontrées avec Coq, nous ne donnons ici que des intuitions de preuve et nous renvoyons le lecteur à l'annexe C pour les détails.

1. Soient deux relations $R_1 \subseteq E \times F$ et $R_2 \subseteq F \times G$, la relation $R_1 \,.\, R_2 \subseteq E \times G$ est définie par $(x, y) \in R_1 \,.\, R_2$ si, et seulement si, il existe $z \in F$ tel que $(x, z) \in R_1$ et $(z, y) \in R_2$.

Proposition 6 (Correction). *Tous les flux engendrés par une exécution observable ω appartiennent à $Flows_\omega$.*

$$\forall e \in \mathbf{E}\ \forall \theta \in \Theta\ \theta_{init} \xhookrightarrow{e_\mathcal{X}} \theta \Rightarrow \forall c \in \mathcal{C}\ \theta(c) \subseteq \bigcup_{(c',c) \in Flows_{e_\mathcal{O}}} \theta_{init}(c')$$

Amorce de preuve. Par induction sur $\lg(e)$. Il suffit de montrer que si une exécution cachée existe, alors, par les conditions de causalité (6.1), il existe nécessairement une séquence d'événements observables ayant activés les flux de l'exécution cachés. Par construction, ces flux sont donc dans $Flows_\omega$. □

La propriété de correction est écrite comme suit en Coq. S'il existe une configuration concrète compatible avec une exécution observable donnée et apportant un certain tag t à un conteneur c, alors il existe un flux d'un conteneur c' qui portait ce tag t dans la configuration initiale vers c. En d'autres termes, toutes les exécutions concrètes sont prises en compte dans la relation $Flows$.

```
Program Theorem soundness :
∀o x e, e ≠ [] → compatible o x e →
∀theta, ConcreteExecution initConfiguration x theta →
∀(c:Container) (t:Tag), theta c t →
 (exists c', Flows o c' c ∧ initConfiguration c' t).
```

Proposition 7 (Plus petite surapproximation / Complétude). *Tous les flux de $Flows_\omega$ sont générés par au moins une exécution cachée compatible avec ω.*

$$\forall \omega \in \mathbf{O}\ \forall c, c' \in \mathcal{C}$$
$$(c, c') \in Flows_\omega \Rightarrow \exists x \in \mathbf{X} \left(\omega \vdash x \wedge \forall \theta \in \Theta\ \theta_{init} \xhookrightarrow{x} \theta \Rightarrow \theta_{init}(c) \subseteq \theta(c') \right)$$

Amorce de preuve. Par induction sur $\lg(\omega)$. Supposons que l'on ait $(c, c') \in Flows_{\omega[:n]}$ un flux composé $(c = c_1, c_2), (c_2, c_3), \ldots, (c_{m-1}, c_m = c')$. Alors, par définition, il existe $i \leq m$ tels que $(c_1, c_i) \in Flows_{\omega[:n-1]}$ et $(c_i, c_m) \in Enabled^*_{\omega[:n]}$. Par l'hypothèse d'induction il existe $x \vdash \omega[:n-1]$ propageant les tags de c_1 à c_i. En concaténant x avec les exécutions des flux $(c_i, c_{i+1}), \ldots, (c_{m-1}, c_m)$ (qui sont activés mais pas encore désactivés dans $\omega[:n]$) dans cet ordre, on obtient une exécution cachée $x' \vdash \omega[:n]$ causant la propagation de tags de $c = c_1$ à $c_m = c'$ via c_i. □

La proposition suivante est l'écriture en Coq du théorème indiquant qu'il n'y a pas de plus petite relation que $Flows$ comportant tous les flux possibles. C'est-à-dire que si un flux de c_1 vers c_2 existe dans $Flows$, il existe nécessairement une exécution observable compatible transférant les tags de c_1 vers c_2.

```
Program Theorem completeness :
∀(o:ObsExecution) (c1 c2:Container),
 Flows o c1 c2 →
∃(x:HidExecution) (e:Execution),
 (compatible o x e ∧
  (forall theta, ConcreteExecution initConfiguration x theta →
   (forall t, initConfiguration c1 t → theta c2 t))).
```

TABLE 6.2 – Exemple d'application de la nouvelle propagation de teintes

(a) $Enabled_{\omega[:n]}$, $Enabled^*_{\omega[:n]}$ et $Flows_{\omega[:n]}$

n	$Enabled_{\omega[:n]}$	$Enabled^*_{\omega[:n]}$	$Flows_{\omega[:n]}$
0		(src, src), (se, se), (p, p), (r, r), (d, d)	(src, src), (se, se), (p, p), (r, r), (d, d)
1	(p, r)	(src, src), (se, se), (p, p), (r, r), (d, d), (p, r)	(src, src), (se, se), (p, p), (r, r), (d, d), (p, r)
2	(p, r), (src, se)	(src, src), (se, se), (p, p), (r, r), (d, d), (p, r), (src, se)	(src, src), (se, se), (p, p), (r, r), (d, d), (p, r), (src, se)
3	(se, p), (src, se), (p, src)	(src, src), (se, se), (p, p), (r, r), (d, d), (se, src), (se, p), (src, p), (src, se), (p, se), (p, src)	(src, src), (se, se), (p, p), (r, r), (d, d), (p, src), (p, se), (src, se), (src, p), (se, src), (se, p)
4 5 6		(src, src), (se, se), (p, p), (r, r), (d, d)	(src, src), (se, se), (p, p), (r, r), (d, d), (p, src), (p, se), (src, se), (src, p), (se, src), (se, p)
7	(r, src)	(src, src), (se, se), (p, p), (r, r), (d, d), (r, src)	(src, src), (se, se), (p, p), (r, r), (d, d), (p, src), (p, se), (src, se), (src, p), (se, src), (se, p), (r, src)
8	(r, d)	(src, src), (se, se), (p, p), (r, r), (d, d), (r, d)	(src, src), (se, se), (p, p), (r, r), (d, d), (p, r), (p, d), (src, se), (src, p), (src, r), (src, d), (se, p), (se, r), (se, d)

(b) Calcul de $\bigcup_{(c_1,c_2)\in Flows_{\omega[:n]}} \theta_{init}(c_1)$ (Surapproximation de Rfblare)

n	$\omega[n]$	$\theta(src)$	$\theta(se)$	$\theta(p)$	$\theta(r)$	$\theta(d)$
1	$p \xrightarrow{enable}_{f_1} r$	★	■	□	◎, □	▲
2	$src \xrightarrow{enable}_{f_2} se$	★	■, ★	□	◎, □	▲
3	$src \xrightarrow{disable}_{f_2} se$	★	■, ★	□	◎, □	▲
4	$se \xrightarrow{enable}_{f_3} p$	★	■, ★	□, ■, ★	◎, □, ■, ★	▲
5	$p \xrightarrow{disable}_{f_1} r$	★	■, ★	□, ■, ★	◎, □, ■, ★	▲
6	$se \xrightarrow{enable}_{f_3} p$	★	■, ★	□, ■, ★	◎, □, ■, ★	▲
7	$r \xrightarrow{enable}_{f_4} d$	★	■, ★	□, ■, ★	◎, □, ■, ★	▲, ◎, □, ■, ★
8	$r \xrightarrow{disable}_{f_4} d$	★	■, ★	□, ■, ★	◎, □, ■, ★	▲, ◎, □, ■, ★

6.3 Implémentation et expérimentations

6.4 Conception

Nous avons implémenté notre algorithme de propagation de teinte sous la forme d'un module de sécurité LSM appelé *Rfblare* (pour *race-free Blare*, *Blare* sans condition de concurrence) en reprenant la base de code de *KBlare*. Nous avons utilisé la version 4.7 du noyau officiel, la dernière disponible à la date de début du projet. Nous avons implémenté uniquement la propagation de teintes, et n'avons pas contribué à l'implémentation des politiques de sécurité, ni à la propagation des teintes sur le réseau implémentée par *KBlare*. Nous ne discutons donc pas ces points ici. Les appels système surveillés par *Rfblare* sont listés dans le tableau 6.3. On peut constater dans un premier temps que *Rfblare* s'attache à couvrir un grand nombre de canaux ouverts, certains bien connus comme `read`, `write` et leurs dérivés, d'autres spécifiques à Linux comme `process_vm_readv`, bien que l'on ne puisse pas prétendre à l'exhaustivité.

Conformément à la description formelle de l'algorithme, nous utilisons un crochet LSM comme événement d'activation du flux et un autre comme événement de désactivation. La projection de notre modèle sur le code du noyau n'est pas immédiate et la connaissance acquise au cours du travail présenté dans le chapitre 5 nous a servi ici. Il est intéressant de noter que l'événement de désactivation n'est pas systématiquement nécessaire. En effet, certains appels système ne peuvent pas entrer en condition de concurrence avec d'autres car le conteneur sur lequel ils agissent est verrouillé par ailleurs, ou bien il est créé par l'appel système lui-même. C'est le cas de l'appel système `exec` par exemple. Cet appel remplace la mémoire du processus appelant pour exécuter un nouveau programme. Au moment où les crochets LSM `bprm_set_creds` et `bprm_committing_creds` sont appelés, le fichier est verrouillé et ne peut être la destination de flux. La mémoire du nouveau processus quant à elle est sur le point d'être entièrement remplacée et la nouvelle mémoire n'est pas encore installée. Elle ne peut donc pas encore être la source de flux. Dans le cas de `fork`, la situation est similaire. Le crochet `mm_dup_security` n'est pas placé pour contrôler la création du nouveau processus mais uniquement pour dupliquer la structure de sécurité associée à sa mémoire. Cette duplication est faite à un point où la mémoire nouvellement créée ne peut pas encore être ni la source ni la destination d'un flux ; par conséquent, il n'y a pas de conditions de concurrence possibles. Enfin, `mq_timedsend` and `msgsnd` sont aussi des cas particuliers. Lorsqu'un message est envoyé dans une file de messages, il est d'abord copié dans un tampon de mémoire du noyau. Le crochet LSM est appelé à ce moment, avant d'insérer effectivement le message dans la file. Cependant, comme le message est déjà copié, il ne peut plus être la cible de flux. Ceci est délibéré, et empêche que le processus émetteur manipule le message après sa vérification. Le noyau évite donc déjà la condition de concurrence sur le conteneur d'information (le message) dans ce cas, il est donc inutile pour *Rblare* d'ajouter son propre mécanisme, qui serait redondant.

La majorité des crochets correspondant aux événements d'activation était déjà présents, comme nous l'avons vérifié dans le chapitre 5. Les crochets LSM correspondant aux événements de désactivation ont dus être ajoutés en revanche. Nous nous contentons de deux crochets différents, car conformément à notre modèle, la propagation de flux est en réalité effectuée à l'activation du flux, et la désactivation sert en réalité à maintenir la liste des flux activés. Les crochets ajoutés sont `syscall_before_return` pour les flux discrets et `ptrace_unlink` d'une part pour

`process_vm_readv` et `process_vm_writev` (qui sont des flux discrets) et d'autre part pour `ptrace` (flux continu). `ptrace` est un appel système particulier utilisé principalement pour le débogage et le diagnostic. Il permet à un processus de s'attacher à un autre et de contrôler son exécution. Le processus contrôleur peut en particulier manipuler les registres et la mémoire du processus attaché. En réalité, les manipulations de mémoire réclament un autre appel à `ptrace` même une fois l'attachement effectué, on pourrait donc considérer chacun de ces flux comme des flux discrets. Cependant, `ptrace` ouvre de nombreux canaux ouverts et cachés de communications entre les deux processus, qui dépendent de plus de l'architecture. Il est donc plus commode de considérer `ptrace` comme un flux continu. Pour ce qui est des flux discrets, nous avons utilisé le nouveau crochet `syscall_before_return`, placé juste avant le retour des appels système concernés. Ce crochet sert uniquement à retirer de la liste des flux activés le flux en cours, quel qu'il soit. L'unique exception est `process_vm_readv` et `process_vm_writev`. Ces deux appels système possédaient déjà le crochet `ptrace_access_check` utilisé par ailleurs par `ptrace`. Pour garantir la cohérence de notre liste de flux activés, et par simplicité, nous avons utilisé dans ce cas le même crochet de fin que pour `ptrace`. Le cas de `shmat`, `mmap` et `mprotect` est particulier. Contrairement aux autres flux où *Rfblare* maintient lui-même l'information d'activation et de désactivation, les flux causés par les fichiers projetés en mémoire est géré en interrogeant le noyau lui-même sur les projections actives. Plus précisément, pour chaque fichier, le noyau maintient la liste des mémoires de processus dans lequel il est projeté ; et pour chaque mémoire, on peut également connaître les fichiers qui sont projetés dedans. Le noyau maintient cette connaissance pour son propre usage, en particulier la libération correcte des ressources. Nous disposons donc en permanence d'un graphe précis et à jour, jamais désynchronisé de la réalité, en ce qui concerne les flux dus aux fichiers projetés. Nous utilisons ce graphe pour propager les teintes de manière appropriées. Nous utilisons néanmoins les crochets LSM d'activation du flux afin de propager les teintes dès l'apparition d'une projection. La projection est unidirectionnelle (du fichier vers le processus) si elle est en lecture seule et bidirectionnelle si elle est en lecture–écriture. Comme `mprotect` peut changer les permissions d'une projection de lecture seule à lecture–écriture, il est important de le prendre en compte.

Le cœur de *Rfblare* est une simple table de flux activés, doublement indexée par la source du flux (ce qui forme donc un graphe) et par le processus ayant déclenché le flux (essentiellement afin de permettre la désactivation du flux et la libération des ressources). À chaque nouvelle apparition d'un flux, un parcours du graphe des flux activés est réalisé afin de propager les teintes à tous les conteneurs atteignables. En plus du graphe maintenu par *Rfblare*, nous utilisons aussi le graphe des fichiers projetés, comme décrit plus haut. Le parcours en largeur converge car la fonction calculée est monotone. On peut arrêter le parcours d'un chemin du graphe dès la rencontre d'un nœud possédant déjà les tags de la source du flux.

6.5 Tests

Nous avons testé les attaques effectuées sur *KBlare* et présentées en section 6.1.1 afin de valider le bon fonctionnement de la propagation de teinte. Dans le cas de la première attaque exploitant la condition de concurrence sur les tubes, la séquence d'événements est bien sûr la même que celle présentée dans la figure 6.1 mais *Rfblare* propage correctement les teintes. Le premier événement est l'activation du flux du tube vers le récepteur. Le flux est enregistré comme actif à ce moment puis le récepteur

TABLE 6.3 – Flux surveillés par *Rfblare* et crochets LSM utilisés

Appel système	Activation	Désactivation
Discrete flows		
`read`	`file_permission`[a]	`before_return`[c]
`readv`	`file_permission`[a]	`before_return`[c]
`preadv`	`file_permission`[a]	`before_return`[c]
`pread64`	`file_permission`[a]	`before_return`[c]
`write`	`file_permission`[a]	`before_return`[c]
`writev`	`file_permission`[a]	`before_return`[c]
`pwritev`	`file_permission`[a]	`before_return`[c]
`pwrite64`	`file_permission`[a]	`before_return`[c]
`sendfile`	`file_permission`[a]	`before_return`[c]
`sendfile64`	`file_permission`[a]	`before_return`[c]
`recv`	`socket_recvmsg`[a]	`before_return`[c]
`recvmsg`	`socket_recvmsg`[a]	`before_return`[c]
`recvmmsg`	`socket_recvmsg`[a]	`before_return`[c]
`recvfrom`	`socket_recvmsg`[a]	`before_return`[c]
`send`	`socket_sendmsg`[a]	`before_return`[c]
`sendmsg`	`socket_sendmsg`[a]	`before_return`[c]
`sendmmsg`	`socket_sendmsg`[a]	`before_return`[c]
`sendto`	`socket_sendmsg`[a]	`before_return`[c]
`process_vm_readv`	`ptrace_access_check`[a]	`ptrace_unlink`[c]
`process_vm_writev`	`ptrace_access_check`[a]	`ptrace_unlink`[c]
`migrate_pages`	`task_movememory`[a]	`before_return`[c]
`move_pages`	`task_movememory`[a]	`before_return`[c]
`fork`	`mm_dup_security`[c]	—[d]
`clone`	`mm_dup_security`[c]	—[d]
`execve`	`bprm_set_creds`[a] / `bprm_committing_creds`[a]	—[d]
`execveat`	`bprm_set_creds`[a] / `bprm_committing_creds`[a]	—[d]
`msgrcv`	`mq_store_msg`[a]	`before_return`[c]
`msgsnd`	`msg_msg_alloc_security`[a]	—[d]
`mq_timedreceive`	`mq_store_msg`[b]	`before_return`[c]
`mq_timedsend`	`msg_msg_alloc_security`[b]	—[d]
Flux continus		
`shmat`	`mmap_file`[a]	—[e]
`mmap_pgoff`	`mmap_file`[a]	—[e]
`mmap`	`mmap_file`[a]	—[e]
`ptrace`	`ptrace_access_check`[a]	`ptrace_unlink`[c]
`ptrace`	`ptrace_traceme`[a]	`ptrace_unlink`[c]

[a] - Crochet LSM déjà présent. [b] - Crochet LSM existant et réutilisé pour cet appel système.
[c] - Crochet ajouté par nos soins.
[d] - Pas de crochet LSM nécessaire car cet appel ne peut pas entrer en concurrence avec un autre sur le même conteneur.
[e] - Pas de crochet LSM nécessaire car l'information des flux activés concernant les fichiers projetés est obtenue par un autre moyen, en interrogeant le noyau.

s'endort avant de sortir de l'appel système. Ensuite, le processus émetteur active le flux du fichier source vers sa mémoire. À ce point, le tag du fichier source est transféré à l'émetteur. Comme aucun flux ayant pour source l'émetteur n'est actif, la propagation s'arrête à ce point. Le flux est ensuite désactivé après la lecture du fichier et le flux est retiré de la liste des flux actif. L'événement suivant est l'activation du flux d'écriture dans le tube par l'émetteur. À ce point, le flux du tube vers le récepteur est toujours activé, et par conséquent la teinte de l'émetteur, contenant le tag du fichier, est transmise au tube et au récepteur. Enfin, les flux sont désactivés, et le flux du récepteur au fichier destination a lieu. Le tag initial du fichier source, et celui de tous les conteneurs du chemin de flux, se retrouve dans la teinte du fichier destination. Le contenu du fichier destination, modifié par celui de la source, est donc reflété correctement dans ses teintes, en dépit de la différence entre l'ordre des flux et l'ordre des événements d'activation de ceux-ci.

Nous avons également reproduit la seconde attaque utilisant des fichiers projetés en mémoire et une zone de mémoire partagée en remplacement du tube. Dans cette attaque, nous pouvons faire varier l'ordre dans lequel les projections et la mémoire partagée sont mises en place, toujours en faisant en sorte de copier le contenu du fichier source dans la mémoire partagée, puis de la mémoire partagée dans le fichier destination. Grâce aux structures de données du noyau nous permettant de connaître la correspondance entre les fichiers projetés et les mémoires de processus, *Rfblare* est capable dans tous les cas de propager les teintes correctement.

Nous avons également mené des expérimentations pour évaluer le surcoût en performances entraîné par l'emploi de *Rfblare*, étant donné que son algorithme de propagation de teintes requiert dans la pratique le calcul d'une femeture transitive de graphes. Les mesures que nous pratiquons ne sont pas directement comparables avec d'autres moniteurs de flux car nous ne mesurons pas le coût de la vérification de la légalité des flux, ni de la réaction (arrêt des processus incriminés, levée d'alertes, communications vers un gestionnaire de politique implémenté en espace utilisateur, etc.). En revanche, les mesures caractérisent exactement ce qui différencie *Rfblare* des autres moniteurs de flux, à savoir la propagation. Nous avons établi un cas de test, qui consiste à compiler le noyau Linux 4.7, sur un système où *Rfblare* est installé. Nous avons fait une première compilation pour savoir quels fichiers étaient impliqués dans la compilation de l'image noyau finale, à l'aide de la table des symboles et des informations de débogage du noyau. Nous avons ensuite utilisé cette liste pour attribuer un tag unique à un nombre variable de ces fichiers. Ensuite, nous relançons une compilation pour étudier l'impact du nombre de tags à propager sur la rapidité de la compilation. Les mesures ont été réalisées sur une machine physique, possédant 16 Go de mémoire vive et huit cœurs de 3,2 GHz de deux threads chacun. Chaque mesure a été répliquée au moins trente fois. Nous avons choisi la compilation comme cas de test pour plusieurs raison. En premier lieu, c'est un test reproductible. Deuxièmement, la compilation implique beaucoup d'entrées–sorties et de manipulation de fichiers, ainsi que de créations de processus. Enfin, il est possible de vérifier à grande échelle le bon fontionnement de *Rfblare* en regardant quels tags sont propagés dans l'image finale et les fichiers intermédiaires et en comparant ces informations avec la table des symboles et les informations de débogage. Nos résultats sont présentés dans la table 6.4. On peut remarquer que dans ce test, l'impact de *Rfblare* est imperceptible et non mesurable. De plus anciennes mesures montraient un temps système accru de quelques pourcents, mais les dernières versions de *Rfblare* ne montrent plus de différence par rapport à un noyau sans *Rfblare*. Ces résultats indiquent que dans ce cas, le goulet d'étranglement n'est pas la propagation de teintes mais probablement

les entrées–sorties ou encore la synchronisation entre les processus participant à la compilation. Même si l'expérimentation que nous avons réalisée ne teste pas un pire cas (mais un cas réaliste tout de même), nous avons la confirmation que la propagation de teintes que nous proposons ne semble pas présenter de prime abord un coût prohibitif en termes de temps d'exécution.

6.6 Conclusion

Les travaux décrits dans ce chapitre ont été présentés à la conférence SEFM 2017 (*Software Engineering & Formal Methods*) [34]. L'article [34] a été récompensé du *Best Paper Award*. Nous avons proposé ici essentiellement un nouveau moyen d'effectuer la propagation de teintes dans un module de sécurité Linux s'appuyant sur le *framework* LSM. Contrairement aux précédentes approches dans le domaine, nous sommes en mesure de traiter :

— les flux causés par des processus effectuant des appels système concurrents et portant sur les mêmes conteneurs d'information ;

— le canal d'information constitué par les fichiers projetés en mémoire, y compris les mémoires partagées entre processus.

En réalité, le premier point est une condition nécessaire pour le deuxième. En effet, les flux en mémoire, et les flux continus en règle générale, sont un exemple de flux pouvant entrer en concurrence avec de nombreux autres. Si les paires de processus A et B d'une part, et B et C d'autre part, partagent respectivement une zone de mémoire, alors il est nécessaire que le moniteur de flux d'information agisse comme si A et C partageaient une zone de mémoire également. En réalité, cette mémoire n'existe pas réellement et est seulement une composition des véritables zones de mémoire partagée.

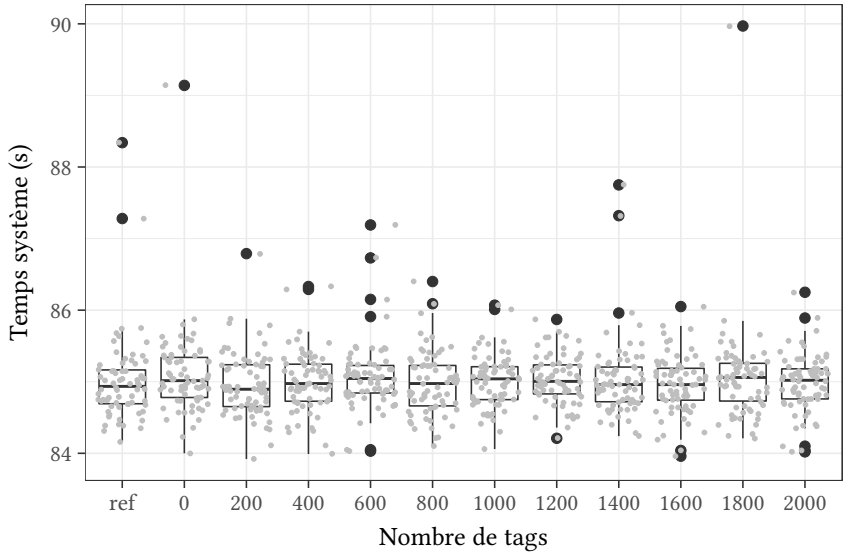
Nous atteignons ces objectifs en ne considérant plus les flux comme des événements atomiques mais des séquences d'activation, exécutions éventuelles et désactivation, avec la particularité que seules les activations et désactivations sont observables par les moniteurs de flux d'information implémentés avec LSM, car elles correspondent à des crochets LSM. Dans le cas particulier des flux causés par les projections en mémoire de fichiers, ce qui comprend les mémoires partagées, on ne cherche pas à détecter l'activation et la désactivation mais on interroge directement le noyau, au moment de réaliser la propagation, pour connaître les flux activés. Notre approche nécessite de changer plusieurs aspects du noyau puisqu'il nous faut des points d'observation de la fin des flux. Dans notre prototype, nous l'avons fait en ajoutant des crochets au *framework* LSM. Si cela se révélait indésirable, il serait possible à la place d'implémenter un mécanisme permettant d'observer tous les retours d'appels système et de considérer ceux-ci comme les points de désactivation des flux discrets. Cette solution ne serait cependant pas aisée à implémenter de manière portable, car le retour de l'espace noyau à l'espace utilisateur est implémenté différemment selon les architectures.

Cette contribution a aussi permis d'exhiber quelques limitations de la contribution exposée au chapitre précédent. Par exemple, dans les appels système de la famille de `splice`, dans le cas d'un flux d'un fichier à un tube, un crochet LSM est bien traversé mais il s'agit du crochet `file_permission`. Le flux qui est donc observé par le moniteur est donc le même que dans le cas de la lecture du fichier, c'est-à-dire un flux du fichier vers la mémoire du processus courant, et non du fichier vers le tube. Ce dernier n'est pas connu du moniteur. Pour effectuer une propagation de teintes correcte dans ce cas, il faut dans le noyau pour lequel est implémenté Rfblare changer le crochet

Table 6.4 – Résultat du micro-*benchmark* de compilation

Nombre de tags	Temps utilisateur (s)	Temps système (s)	Temps écoulé (s)
réf.	1134 ± 1.06	85.02 ± 0.145	211.4 ± 0.42
0	1136 ± 1.03	85.10 ± 0.148	211.9 ± 0.57
200	1136 ± 1.11	84.96 ± 0.108	212.0 ± 0.66
400	1136 ± 1.03	85.00 ± 0.101	212.2 ± 0.57
600	1136 ± 1.08	85.07 ± 0.112	211.5 ± 0.37
800	1135 ± 1.02	84.98 ± 0.107	211.7 ± 0.48
1000	1136 ± 1.09	85.00 ± 0.087	212.1 ± 0.57
1200	1136 ± 0.85	85.03 ± 0.081	211.6 ± 0.52
1400	1137 ± 0.96	85.04 ± 0.131	212.2 ± 0.54
1600	1135 ± 1.10	84.96 ± 0.096	212.1 ± 0.55
1800	1136 ± 0.84	85.06 ± 0.166	212.0 ± 0.41
2000	1136 ± 1.04	84.99 ± 0.095	211.9 ± 0.48

Les temps sont donnés comme une moyenne sur trente mesures au moins, avec leur intervalle de confiance à 95%.
La première colonne identifie le cas de test. La référence est un système identique à celui des autres cas, à ceci près que *Rfblare* n'est pas utilisé. Le reste des cas est identifiés par le nmbre de fichiers sources portant un tag au début de la compilation.
La colonne « Temps utilisateur » donne le temps cumulé passé par le compilateur en-dehors du noyau, dans l'espace utilisateur.
La colonne « Temps système » donne à l'inverse le temps passé dans le noyau, à exécuter des appels système, et donc entre autres, à effectuer la propagation de teintes.
La colonne « Temps écoulé » donne le durée totale de la compilation. Cette valeur est différente de la somme des deux précédentes valeurs car elle ne cumule pas les temps passés par les différents processus et threads de compilation s'exécutant en parallèle.



Les points en gris clair sont les mesures individuelles, les boîtes à moustache donnent les quartiles des mesures, les points en noir sont les valeurs extrêmes.

par un autre, prenant en paramètre le fichier et le tube. Cela ouvre la voie à de futurs travaux, portant sur une sémantique formelle des appels système en termes de flux d'information et une analyse statique de la correspondance entre les crochets LSM traversés dans les appels système et ces flux. Cela rappelle le travail effectué par Jaeger, Edwards et Zhang dans « Consistency analysis of authorization hook placement in the Linux security modules framework » [42] à ceci près que cette nouvelle analyse porterait sur le contrôle de flux d'information et non le contrôle d'accès.

Chapitre 7

Conclusion

Nous avons abordé dans le cadre de cette thèse plusieurs types de problèmes. Un problème d'ingénierie logicielle tout d'abord. Le noyau Linux est riche, à l'heure où nous écrivons ces lignes, de vingt-cinq ans d'histoire mouvementée depuis le jour où Linus Torvalds a fait la publicité sur Usenet de son noyau nouvellement créé. Plusieurs milliers de fichiers, et plusieurs millions de lignes de code, se sont accumulés pour former le noyau tel qu'on peut l'utiliser aujourd'hui, sur de nombreuses architectures et types d'équipements différents, depuis les téléphones Android jusqu'aux supercalculateurs. Nos travaux portant sur l'analyse de propriétés sur le code du noyau, le premier travail de cette thèse a porté sur la compréhension du code. L'expertise acquise durant cette phase est ré-exploitable. Nous avons implémenté le projet Kayrebt, une collection de logiciels comprenant notamment deux greffons GCC et une interface graphique permettant de visualiser la collection de graphes de flot de contrôle extraits d'une base de code.

Le second problème auquel nous nous sommes attaqué est un problème de sémantique. Il porte sur le moyen d'analyser le code des appels système produisant des flux d'informations pour vérifier la bonne position des crochets LSM. Nous avons pour cela défini un modèle du code, basé sur les graphes de flot de contrôle tels qu'extraits du code par Kayrebt::Extractor. Nous avons muni ces graphes d'une sémantique formelle, que nous avons prouvée correcte vis-à-vis d'hypothèses communes et simples à propos de la sémantique concrète du code. L'analyse que nous avons conçue nous a permis de déterminer précisément à quel point le *framework* LSM permet de suivre les flux d'informations. Nous avons conclu que ce *framework* est en effet adapté au suivi de flux d'information à quelques exceptions près. Notre approche montre tout son intérêt pour décrire précisément quels sont les chemins d'exécutions qui ne sont pas couverts par les crochets LSM. De plus, l'analyse est reproductible, ce qui permet de suivre les évolutions du noyau. Nous avons nous-mêmes analysé les versions 4.3 et 4.7 du noyau Linux, en plus de notre version modifiée d'après les résultats de la première analyse.

Notre troisième contribution nous a donné l'occasion de nous attaquer à un problème algorithmique et d'implémentation dans le noyau Linux. Ayant constaté et démontré le problème qu'ont les moniteurs de flux d'information implémentés avec LSM, nous avons montré que ce problème est dû à des conditions de concurrence, et qu'on peut le rapprocher de problèmes de causalité bien connus. En effet, le problème des conditions de concurrence se pose lorsque deux flux impactant le même conteneur d'information ne sont pas observés dans l'ordre causal où ils sont effectués. Cela fait que le flux indirect composé des deux flux n'est pas pris en compte car du point de

vue du moniteur de flux, ils n'a pas pu logiquement avoir lieu. Nous avons conçu et prouvé un nouvel algorithme de propagation de teintes permettant de résoudre ce problème en calculant une surapproximation des teintes dans le cas où plusieurs ordres de génération des flux sont possibles et que le moniteur ne peut pas connaître l'ordre correct. Nous avons prouvé la correction de cet algorithme à l'aide de Coq. Nous l'avons ensuite implémenté dans le noyau. Il y a une sous-contribution à ce point. Pouvoir gérer les conditions de concurrence est une condition nécessaire pour pouvoir traiter les flux continus et nous avons de plus montré comment les informations déjà maintenues par le noyau peuvent être exploitées par les moniteurs de flux pour traiter le cas des flux causés par les projections de fichiers et les mémoires partagées. Notre implémentation, Rfblare, est donc le premier moniteur de flux, à notre connaissance, à gérer correctement le canal ouvert constitué par les projections de fichiers et les mémoires partagées.

À la question de savoir si le *framework* LSM est bien adapté à l'implémentation du suivi de flux d'information, nous avons donc apporté une réponse ambigüe. Dans le chapitre 5, avons montré que les crochets LSM sont des points d'observation appropriés en général, moyennant quelques modifications. En revanche, lorsque nous avons proposé notre propre moniteur dans le chapitre 6, nous avons modifié la structure du *framework* pour ajouter des crochets à la fin des appels système. Les modifications ne sont pas très importantes en termes de nombre de lignes de code modifiées mais elles mettent en évidence les limites de la flexibilité et de la généricité de LSM. Des travaux ambitieux prolongeant cette thèse mériteraient d'être approfondis, à commencer par une sémantique des appels système en termes de flux d'information, aussi formelle que possible. En effet, nous avons évoqué dès le chapitre 2 que le noyau est comme une machine à faire des flux, dont les appels système sont les instructions. Cependant, contrairement à un processeur ou un langage, il n'y a pas de sémantique claire et définitive des appels système, en particulier de ceux qui sont propres à Linux, mais seulement un ensemble de normes et de règles exprimées en langage naturel et présentées dans la documentation du noyau. Cette sémantique augmenterait la confiance que l'on peut avoir dans les mécanismes de sécurité implémentés avec LSM ainsi que la portée des analyses comme celles que nous avons développées.

Enfin, nous avons produit une contribution annexe dans cette thèse qui possède, à nos yeux, une grande importance : l'usage du compilateur pour la production d'analyses statiques et de visualisations. Qu'il s'agisse de comprendre l'organisation d'une base de code, d'explorer les chemins d'exécutions d'une fonction, d'implémenter une analyse statique, de déboguer un morceau de code ou encore de comprendre le fonctionnement d'une phase particulière du processus de compilation, les artefacts de compilation et les structures de données du compilateur peuvent être mises à profit.

Un problème que nous ne pouvons cependant pas traiter dans le cadre d'une seule thèse, quel que soit le temps qu'on lui accorderait, est celui de l'adoption du contrôle de flux d'information comme moyen de sécurité dans les systèmes d'exploitation. En effet, à l'heure actuelle, le contrôle de flux est bien moins prévalent que le contrôle d'accès. Dans le noyau Linux *vanilla*, on dénombre quatre modules de sécurité implémentant un contrôle d'accès obligatoire : AppArmor, SELinux, Smack et Tomoyo, mais aucun module de contrôle de flux. Dans d'autres domaines comme l'analyse dynamique de logiciels malveillants, le contrôle de flux a cependant déjà démontré son intérêt. Les problèmes que nous avons traités sont donc plus que jamais d'actualité.

Annexe A

Liste des crochets LSM dans la version 4.7 du noyau

A.1 Crochets présents dans le noyau officiel

Le tableau ci-après liste exhaustivement l'ensemble des crochets LSM disponibles dans la version 4.7 du noyau Linux *vanilla*, groupés par les structures de données qu'ils protègent. Les crochets soulignés sont ceux utilisés dans l'implémentation de Rfblare.

A.1.1 Contrôle des requêtes faites au *binder* d'Android

`binder_set_context_mgr` Appelé pour contrôler le changement de gestionnaire du *binder*.

`binder_transaction` Appelé pour contrôler un appel via le *binder* d'un processus à un autre.

`binder_transfer_binder` Appelé pour contrôler la transmission d'une référence sur le *binder* d'un processus à un autre.

`binder_transfer_file` Appelé pour contrôler la transmission d'un fichier via le *binder* d'un processus à un autre.

A.1.2 Contrôle des appels à `ptrace`

`ptrace_access_check` Appelé lorsqu'un thread effectue un appel système `ptrace` sur un thread d'un autre processus pour contrôler l'opération demandée.

`ptrace_traceme` Appelé lorsqu'un thread demande à un thread d'un autre processus de le tracer pour contrôler cette opération.

A.1.3 Gestion et requête des autorisations des threads (`capabilities`)

`capget` Appelé lorsqu'un thread appelle `capget` pour que le module retourne les capacités d'un thread.

`capset` Appelé lorsqu'un thread appelle `capset` pour que le module modifie les capacités d'un thread.

`capable` Appelé lors de la vérification qu'un thread possède une certaine capacité, pour que le module retourne la réponse.

A.1.4 Gestion des quotas de ressources

`quotactl` Appelé pour contrôler l'appel aux fonctions de gestion des quotas de ressources pour les utilisateur et leurs processus.

`quota_on` Appelé pour contrôler le respect des quotas sur un fichier.

A.1.5 Contrôle de l'accès au journal du noyau

`syslog` Appelé pour contrôler l'accès par un thread au journal du noyau (tampon circulaire où le noyau imprime ses journaux).

A.1.6 Contrôle de la manipulation de l'horloge du système

`settime` Appelé pour contrôler la modification par un thread de l'heure et date courante du système.

A.1.7 Contrôle de l'allocation d'une nouvelle projection en mémoire

`vm_enough_memory` Appelé à divers endroits du noyau lorsqu'une nouvelle zone de mémoire virtuelle doit être allouée pour un processus, afin de contrôler la consommation mémoire.

A.1.8 Contrôle et gestion du lancement d'un nouvel exécutable

`bprm_set_creds` Appelé lors du chargement d'un nouveau programme (avec l'appel système `execve`) pour initialiser les permissions du processus en fonction du programme.

`bprm_check_security` Appelé juste avant la recherche d'un interpréteur ou du programme approprié pour un certain type de binaire lors du chargement d'un nouveau programme, pour contrôler l'opération en cours ou mettre à jour les permissions du processus.

`bprm_secureexec` Appelé lors du chargement d'un programme pour savoir si une exécution sécurisée est requise (ce qui change le comportement de la bibliothèque standard, entre autres choses).

`bprm_committing_creds` Appelé à la fin du chargement d'un nouveau programme pour préparer le processus juste avant que ses permissions ne soient mises à jour avec celles préparées plus tôt par les crochets précédents.

`bprm_committed_creds` Appelé à la fin du chargement d'un nouveau programme, une fois que les permissions du processus sont prêtes pour préparer le processus à son exécution avec ses nouvelles permissions.

A.1.9 Gestion des structures de sécurité des superblocs du système de fichiers virtuel

`sb_alloc_security` Appelé lors de l'initialisation d'un superbloc (par exemple lors du montage d'un système de fichiers) pour allouer et peupler sa structure de sécurité.

`sb_free_security` Appelé lors de la destruction d'un superbloc (par exemple lors du démontage d'un système de fichiers) pour désallouer sa structure de sécurité.

`sb_set_mnt_opts` Appelé lors de la mise à jour des options de sécurité d'un superbloc.

`sb_clone_mnt_opts` Appelé lorsque des options de sécurité doivent être copiées d'un superbloc à un autre.

`sb_parse_opts_str` Appelé pour initialiser les options de sécurité d'un superbloc à partir d'une chaîne de caractère.

`sb_copy_data` Appelé pour copier les options de montage d'un superbloc et les modifier avant qu'elles ne soient passées au système de fichier pour montage.

`sb_show_options` Appelé pour sérialiser les options de montage d'un superbloc dans une chaîne de caractères.

A.1.10 Contrôle des opérations des superblocks du système de fichiers virtuel

`sb_remount` Appelé pour contrôler une opération de remontage d'un système de fichiers.

`sb_mount` Appelé pour contrôler une opération de montage d'un système de fichiers. La fonction attachée au crochet LSM peut vérifier notamment le point de montage et les options demandées.

`sb_umount` Appelé pour contrôler une opération de démontage d'un système de fichiers.

`sb_kern_mount` Appelé juste avant l'initialisation du superbloc, pendant le montage, pour initialiser la structure de sécurité et contrôler si le montage peut se poursuivre.

`sb_statfs` Appelé pour contrôler la récupération de statistiques sur le système de fichiers du superbloc.

`sb_pivotroot` Appelé pour contrôler une opération de changement général de racine du système de fichiers, et éventuellement mettre à jour certaines structures de sécurité au passage.

A.1.11 Gestion des champs de sécurité des entrées de répertoire

`dentry_init_security` Appelé pour peupler le champ de sécurité d'une entrée de répertoire qui n'a pas encore d'i-nœud associé. Ce crochet est utile pour les systèmes de fichiers NFS.

`d_instantiate` Appelé pour initialiser le champ de sécurité d'une entrée de répertoire en train d'être installée dans le cache.

A.1.12 Contrôle des opérations sur les chemins du système de fichiers virtuel

Compilation optionnelle contrôlée par l'option `CONFIG_SECURITY_PATH`.

`path_unlink` Appelé pour contrôler une opération de suppression d'un fichier (appel système `unlink`)

`path_mkdir` Appelé pour contrôler une opération de création d'un répertoire (appel système `mkdir`)

`path_rmdir` Appelé pour contrôler une opération de suppression d'un répertoire (appel système `rmdir`)

`path_mknod` Appelé pour contrôler la création d'un fichier spécial ou d'un tuyau (appel système `mknod`)

`path_truncate` Appelé pour contrôler la troncature ou l'extension d'un fichier (appel système `truncate`)

`path_symlink` Appelé pour contrôler la création d'un lien symbolique (appel système `symlink`)

`path_link` Appelé pour contrôler la création d'un lien dur (appel système `link`)

`path_rename` Appelé pour contrôler le renommage d'un fichier (appel système `rename`)

`path_chmod` Appelé pour contrôler le changement de mode d'accès discrétionnaire d'un fichier (appel système `chmod`)

`path_chown` Appelé pour contrôler le changement de propriétaire ou groupe propriétaire d'un fichier (appel système `chown`)

`path_chroot` Appelé pour contrôler le confinement d'un processus dans une nouvelle racine de système de fichiers (appel système `chroot`)

A.1.13 Gestion des champs de sécurité des i-nœuds du système de fichiers virtuel

`inode_alloc_security` Appelé lorsqu'un i-nœud est créé pour allouer sa structure de sécurité.

`inode_free_security` Appelé lorsqu'un i-nœud est détruit pour désallouer sa structure de sécurité.

`inode_init_security` Appelé pour installer les attributs étendus de sécurité dans un tout nouvel i-nœud.

A.1.14 Contrôle des opérations sur les i-nœuds du système de fichiers virtuel

`inode_create` Appelé pour contrôler la création d'un nouveau fichier simple.

`inode_link` Appelé pour contrôler la création d'un lien dur vers un fichier.

`inode_unlink` Appelé pour contrôler la suppression d'un lien vers un fichier.

`inode_symlink` Appelé pour contrôler la création d'un lien symbolique vers un fichier.

`inode_mkdir` Appelé pour contrôler la création d'un répertoire.

`inode_rmdir` Appelé pour contrôler la suppression d'un répertoire.

`inode_mknod` Appelé pour contrôler la création d'un fichier spécial ou un tuyau.

`inode_rename` Appelé pour contrôler le renommage d'un fichier.

`inode_readlink` Appelé pour contrôler la lecture d'un lien symbolique pour en connaître la destination.

`inode_follow_link` Appelé pour contrôler le déréférencement d'un lien symbolique pour le suivre dans le déchiffrage d'un chemin.

`inode_permission` Appelé pour contrôler toutes les opérations sur les i-nœuds autres que celles relatives à la création ou destruction (par exemple, l'ouverture d'une fichier).

`inode_setattr` Appelé pour contrôler la modification des attributs d'un i-nœud (comme la date de dernier accès par exemple) ;

`inode_getattr` Appelé pour contrôler la lecture des attributs d'un i-nœud.

`inode_setxattr` Appelé pour contrôler la modification des attributs étendus d'un i-nœud, dont les attributs de sécurité. La fonction attachée à ce crochet n'est pas censée écrire elle-même les attributs de sécurité mais seulement vérifier la validité du nom et de la valeur de l'attribut.

`inode_post_setxattr` Appelé pour mettre à jour les structures de sécurité de l'i-nœud et de l'entrée de répertoire après la modification des attributs étendus d'un i-nœud.

`inode_getxattr` Appelé pour contrôler la lecture d'un attribut étendu d'un i-nœud (qui peut être un attribut de sécurité).

`inode_listxattr` Appelé pour lister les attributs étendus d'un i-nœud.

`inode_removexattr` Appelé pour contrôler la suppression d'un attribut étendu d'un i-nœud.

`inode_need_killpriv` Appelé lorsqu'un i-nœud est modifié pour savoir s'il est nécessaire de supprimer le bit `setuid` et autres statuts spéciaux en conséquence.

`inode_killpriv` Appelé lorsque le bit `setuid` d'un i-nœud est supprimé afin que le module de sécurité adapte à son tour le contexte de sécurité de l'i-nœud en conséquence.

`inode_getsecurity` Appelé pour sérialiser un attribut de sécurité de l'i-nœud dans une chaîne de caractères.

`inode_setsecurity` Appelé pour contrôler et effectuer la modification d'un attribut de sécurité d'un i-nœud, lorsque le système de fichiers ne gère pas les attributs étendus. Si le système de fichiers les gère, la paire de crochets `inode_setxattr` et `inode_post_setxattr` est utilisée à la place.

`inode_listsecurity` Appelé pour lister les attributs de sécurité d'un i-nœud dans une chaîne de caractères, lorsque le système de fichiers ne gère pas les attributs étendus. Si le système de fichiers les gère, `inode_listxattr` est utilisé à la place.

`inode_getsecid` Appelé pour retourner l'identifiant de sécurité d'un i-nœud.

`inode_notifysecctx` Appelé pour indiquer au module de sécurité quel est le contexte de sécurité attendu pour un i-nœud. Ce crochet est utilisé par NFS pour construire en différé ses i-nœuds correspondants à des fichiers stockés sur un serveur distant.

`inode_setsecctx` Appelé pour positionner le contexte de sécurité d'un i-nœud, et répercuter ce changement dans les attributs étendus. Ce crochet est utilisé par NFS.

`inode_getsecctx` Appelé pour récupérer le contexte de sécurité d'un i-nœud.

A.1.15 Gestion des champs de sécurité des descripteurs de fichiers

`file_alloc_security` Appelé pour allouer et peupler la structure de sécurité d'un descripteur de fichier.

`file_free_security` Appelé pour désallouer la structure de sécurité d'un descripteur de fichier.

`file_set_fowner` Appelé pour positionner la valeur du propriétaire d'un fichier, pour que la fonction associée au crochet `file_send_sigiotask` puisse l'utiliser.

`file_open` Appelé pour stocker les permissions à l'ouverture du fichier, pour pouvoir mettre en cache les décisions de sécurité.

A.1.16 Contrôle des opérations sur les descripteurs de fichiers

`file_permission` Appelé pour contrôler une opération sur un descripteur de fichier, essentiellement `read` et `write` ou leurs dérivés.

`file_ioctl` Appelé pour contrôler une opération `ioctl` sur un fichier.

`file_mprotect` Appelé pour contrôler le changement de la protection de la projection en mémoire d'un fichier.

`file_lock` Appelé pour contrôler l'usage du verrouillage d'un fichier.

`file_fcntl` Appelé pour contrôler une opération `fcntl` sur un fichier.

`file_send_sigiotask` Appelé pour vérifier que le propriétaire d'un fichier est en mesure d'envoyer le signal `SIGIO` ou `SIGURG`. Ces signaux sont utilisés sur les sockets pour signaler au processus propriétaire que la socket a terminé une opération d'entrée-sortie ou est prête pour une nouvelle opération.

`file_receive` Appelé pour contrôler la réception d'un descripteur de fichier depuis un autre processus via une socket.

A.1.17 Contrôle de la création d'une nouvelle projection en mémoire

`vm_enough_memory` Appelé pour contrôler l'allocation d'une nouvelle zone de mémoire au sein de l'espace d'adressage d'un processus, au cas où le module de sécurité souhaite appliquer des quotas.

`mmap_addr` Appelé pour contrôler la création d'une nouvelle projection en mémoire à une adresse donnée.

`mmap_file` Appelé pour contrôler la projection en mémoire d'un fichier régulier existant.

A.1.18 Gestion des champs de sécurité des threads

`task_free` Appelé à la destruction d'un thread pour désallouer la mémoire associée.

`cred_alloc_blank` Appelé pour allouer une structure de sécurité de processus vierge.

`cred_free` Appelé pour désallouer une structure de sécurité de processus.

`cred_prepare` Allouer pour initialiser une structure de sécurité de processus à partir d'une autre, en la copiant.

`cred_transfer` Appelé pour initialiser une structure de sécurité de processus à partir d'une autre, en transférant le contenu de la structure.

`task_fix_setuid` Appelé pour mettre à jour les structures de sécurité associées à un thread après un changement d'UID, de GID, etc.

`task_to_inode` Appelé pour définir la structure de sécurité d'un i-nœud créé d'après un thread, en particulier pour les fichiers spéciaux décrivant le thread dans `/proc/<pid>`.

A.1.19 Contrôle des opérations impliquant des threads ou processus

`task_create` Appelé pour contrôler la création d'un thread.

`task_setpgid` Appelé pour contrôler le changement de numéro de groupe d'un thread.

`task_getpgid` Appelé pour contrôler la consultation du numéro de groupe d'un thread.

`task_getsid` Appelé pour contrôler la consultation du numéro de session d'un thread.

`task_getsecid` Appelé pour récupérer l'identifiant de sécurité d'un thread.

`task_setnice` Appelé pour contrôler le changement de *niceness* (priorité) d'un processus.

`task_setioprio` Appelé pour contrôler le changement de valeur `ioprio` d'un thread.

`task_getioprio` Appelé pour contrôler la consultation de valeur `ioprio` d'un thread.

`task_setrlimit` Appelé pour contrôler le changement de limite d'une ressource d'un processus.

`task_setscheduler` Appelé pour contrôler le changement de la politique d'ordonnancement s'appliquant à un thread.

`task_getscheduler` Appelé pour consulter la politique d'ordonnancement applicable à un thread.

`task_movememory` Appelé pour contrôler le déplacement de pages d'un nœud NUMA à un autre, et le cas échéant, d'un processus à un autre (par les appels système `move_pages` et `migrate_pages`).

`task_kill` Appelé pour contrôler l'envoi de signal par un processus.

`task_wait` Appelé pour contrôler l'attente d'un évènement d'un processus fils.

`task_prctl` Appelé pour contrôler une opération `prctl`.

A.1.20 Manipulation des attributs de sécurité des processus

`getprocattr` Appelé pour sérialiser sous la forme d'une chaîne de caractères la structure de sécurité d'un thread et contrôler sa récupération.

`setprocattr` Appelé pour contrôler la modification de la structure de sécurité d'un thread et l'effectuer (en extrayant la valeur d'une chaîne de caractères).

A.1.21 Contrôle et gestion des tâches réalisées par un thread du noyau

`kernel_act_as` Appelé pour changer le contexte de sécurité d'un thread du noyau.

`kernel_create_files_as` Appelé pour attribuer à un thread du noyau le contexte de sécurité d'un i-nœud spécifique.

`kernel_read_file` Appelé pour contrôler la lecture par un thread du noyau d'un fichier spécifié par l'espace utilisateur.

`kernel_post_read_file` Appelé pour contrôler le contenu d'un fichier spécifié par l'espace utilisateur lu par un thread du noyau.

`kernel_module_request` Appelé pour contrôler le chargement d'un module via un appel à `modprobe` (en espace utilisateur).

A.1.22 Gestion des champs de sécurité des IPC System V

`msg_msg_alloc_security` Appelé pour allouer la structure de sécurité d'un message destiné à être envoyé via une file de messages (POSIX ou System V).

`msg_msg_free_security` Appelé pour désallouer la structure de sécurité d'un message.

`msg_queue_alloc_security` Appelé pour allouer la structure de sécurité d'une file de messages System V.

`msg_queue_free_security` Appelé pour désallouer la structure de sécurité d'une file de messages System V.

`shm_alloc_security` Appelé pour allouer la structure de sécurité d'une mémoire partagée System V.

`shm_free_security` Appelé pour désallouer la structure de sécurité d'une mémoire partagée System V.

`sem_alloc_security` Appelé pour allouer la structure de sécurité d'un groupe de sémaphores System V.

`sem_free_security` Appelé pour désallouer la structure de sécurité d'un groupe de sémaphores System V.

A.1.23 Contrôle des IPC System V

`ipc_permission` Appelé pour contrôler l'appel à une opération sur une des IPC System V.

`ipc_getsecid` Appelé pour récupérer le contexte de sécurité d'une IPC System V.

`msg_queue_associate` Appelé pour contrôler la récupération de l'identifiant d'une file de messages System V pré-existante.

`msg_queue_msgctl` Appelé pour contrôler une opération portant sur une file de messages System V.

`msg_queue_msgsnd` Appelé pour contrôler l'envoi un message via une file de messages System V.

`msg_queue_msgrcv` Appelé pour contrôler la réception d'un message depuis une file de message System V.

`shm_associate` Appelé pour contrôler la récupération de l'identifiant d'une mémoire partagée System V.

`shm_shmctl` Appelé pour contrôler une opération `shmctl` de manipulation d'une mémoire partagée System V.

`shm_shmat` Appelé pour contrôler l'attachement d'une mémoire partagée System V dans l'espace d'adressage d'un processus.

`sem_associate` Appelé pour contrôler la récupération d'un identifiant de groupe de sémaphore System V.

`sem_semctl` Appelé pour contrôler une opération `semctl` de manipulation d'un groupe de sémaphores System V.

`sem_semop` Appelé pour contrôler une opération sur les membres d'un groupe de sémaphores System V.

A.1.24 Gestion des champs de sécurité des messages Netlink et contrôle de leur émission

`netlink_send` Appelé lors de l'émission d'un message Netlink pour contrôler la transmission et pour attacher une valeur de sécurité au message.

A.1.25 Gestion des contextes de sécurité

`ismaclabel` Appelé pour vérifier si la valeur d'un buffer correspond à un contexte de sécurité valide.

`secid_to_secctx` Appelé pour convertir un identifiant (sans doute opaque) en contexte de sécurité complet.

`secctx_to_secid` Appelé pour obtenir un identifiant de sécurité depuis un contexte de sécurité.

`release_secctx` Appelé pour désallouer un contexte de sécurité.

`inode_invalidate_secctx` Appelé pour invalider le contexte de sécurité d'un i-nœud et forcer sa revalidation depuis les attributs étendus ou autre.

A.1.26 Contrôle des opérations sur les sockets UNIX

Compilation conditionnelle contrôlée par `CONFIG_SECURITY_NETWORK`.

`unix_stream_connect` Appelé pour contrôler l'établissement d'une connexion de type flux entre deux sockets de domaine UNIX.

`unix_may_send` Appelé pour contrôler la connexion et l'émission de datagrammes sur une socket de type UNIX.

A.1.27 Contrôle des opérations sur les sockets

Compilation conditionnelle contrôlée par `CONFIG_SECURITY_NETWORK`.

`socket_create` Appelé pour contrôler la création d'une socket.

`socket_post_create` Appelé après la création d'une socket pour ajouter des attributs de sécurité supplémentaires à la socket. Les sockets utilisent comme structure de sécurité celle de leur i-nœud et une structure de sécurité dédiée.

`socket_bind` Appelé pour contrôler l'attachement d'une socket à une adresse.

`socket_connect` Appelé pour contrôler la connexion à une socket.

`socket_listen` Appelé pour contrôler l'attachement d'une socket à un port et le début de l'écoute.

`socket_accept` Appelé pour contrôler l'acceptation d'une connexion à une socket.

`socket_sendmsg` Appelé pour contrôler l'émission d'un message sur une socket.

`socket_recvmsg` Appelé pour contrôler la réception d'un message sur une socket.

`socket_getsockname` Appelé pour contrôler la récupération de l'adresse locale de la socket.

`socket_getpeername` Appelé pour contrôler la récupération de l'adresse du pair distant de la socket.

`socket_getsockopt` Appelé pour contrôler la récupération des options positionnées sur la socket.

`socket_setsockopt` Appelé pour contrôler la manipulation des options de la socket.

`socket_shutdown` Appelé pour contrôler la fermeture d'une socket.

`socket_sock_rcv_skb` Appelé pour contrôler la réception d'un paquet sur une socket de domaine INET avant de le faire passer dans un filtre réseau.

A.1.28 Gestion des champs de sécurité des sockets

Compilation conditionnelle contrôlée par `CONFIG_SECURITY_NETWORK`.

`socket_getpeersec_stream` Appelé pour récupérer (et en contrôler la récupération) la structure de sécurité associée au pair distant d'une socket en mode flux (INET, utilisant TCP, ou UNIX).

`socket_getpeersec_dgram` Appelé pour récupérer (et en contrôler la récupération) la structure de sécurité associée à un message d'une socket en mode datagramme (INET, donc utilisant UDP).

`sk_alloc_security` Appelé pour allouer une structure de sécurité spécifique aux sockets.

`sk_free_security` Appelé pour désallouer la structure de sécurité spécifique aux sockets.

`sk_clone_security` Appelé pour copier une structure de sécurité spécifique aux sockets.

`sk_getsecid` Appelé pour récupérer l'identifiant de sécurité associée à une socket.

`sock_graft` Appelé pour positionner l'identifiant de sécurité de l'i-nœud de la socket à l'identifiant de sécurité de la socket elle-même.

`inet_conn_request` Appelé lors de la réception d'une demande de connexion pour mettre à jour l'identifiant de sécurité de la demande de connexion d'après celui de la socket du serveur et de l'identifiant de sécurité du pair distant.

`inet_csk_clone` Appelé à l'établissement d'une session pour mettre à jour l'identifiant de sécurité de la socket fille d'après l'identifiant de sécurité de la demande de connexion.

`inet_conn_established` Appelé après l'établissement d'une session pour mettre à jour l'identifiant de sécurité associé au pair distant dans la socket (du côté du serveur), d'après l'identifiant de sécurité associé au paquet reçu.

`secmark_relabel_packet` Appeler pour contrôler une opération de ré-étiquetage d'un paquet par un processus utilisateur.

`secmark_refcount_inc` Appelé pour signaler au module de sécurité qu'une règle de ré-étiquettage supplémentaire a été chargé.

`secmark_refcount_dec` Appelé pour signaler au module de sécurité qu'une règle de ré-étiquettage a été déchargée.

`req_classify_flow` Appelé pour mettre à jour l'identifiant de sécurité d'un flux internet d'après l'identifiant de sécurité de la demande de connexion.

A.1.29 Gestion des champs de sécurité des interfaces réseaux virtuelles

Compilation conditionnelle contrôlée par `CONFIG_SECURITY_NETWORK`.

`tun_dev_alloc_security` Appelé pour allouer une structure de sécurité pour une interface TUN.

`tun_dev_free_security` Appelé pour désallouer la structure de sécurité d'une interface TUN.

A.1.30 Contrôle des opérations sur les interfaces réseaux virtuelles

Compilation conditionnelle contrôlée par `CONFIG_SECURITY_NETWORK`.

`tun_dev_create` Appelé pour contrôler la création d'une nouvelle interface TUN.

`tun_dev_attach_queue` Appelé pour contrôler une opération d'attachement demandée par le processus courant à une interface TUN.

`tun_dev_attach` Appelé pour mettre à jour le contexte de sécurité associé à une socket lorsqu'elle est attachée à une interface TUN.

`tun_dev_open` Appelé pour contrôler une opération d'ouverture d'une interface TUN.

A.1.31 Gestion des champs de sécurité XFRM

Compilation conditionnelle contrôlée par `CONFIG_SECURITY_NETWORK_XFRM`.

XFRM est un framework de transformation des paquets réseaux, il est utilisé pour implémenté IPSec.

`xfrm_policy_alloc_security` Appelé pour contrôler la création d'une nouvelle politique dans la base de politiques et lui allouer une structure de sécurité.

`xfrm_policy_clone_security` Appelé pour allouer une nouvelle structure de sécurité pour une politique et l'initialiser à partir d'une autre.

`xfrm_policy_free_security` Appelé pour désallouer une structure de sécurité de politique.

`xfrm_policy_delete_security` Appelé pour contrôler la destruction d'une politique de la base.

`xfrm_state_alloc` Appelé pour contrôler la création d'une nouvelle association dans la base utilisé par XFRM et allouer la structure de sécurité correspondante. La structure est initialisée à partir d'un contexte fabriqué par un programme en espace utilisateur approprié.

`xfrm_state_alloc_acquire` Similaire à `xfrm_state_alloc` mais le contexte de sécurité avec lequel la nouvelle structure doit être initialisée est spécifié par un identifiant de sécurité.

`xfrm_state_free_security` Appelé pour désallouer la structure de sécurité d'une association.

`xfrm_state_delete_security` Appelé pour contrôler la destruction d'une association.

A.1.32 Contrôle des opérations sur les politiques XFRM

Compilation conditionnelle contrôlée par `CONFIG_SECURITY_NETWORK_XFRM`.

`xfrm_policy_lookup` Appelé pour contrôler l'application de politiques XFRM sur un flux réseau.

`xfrm_state_pol_flow_match` Appelé pour savoir si une politique donnée s'applique à un flux réseau.

`xfrm_decode_session` Appelé pour contrôler que tous les paquets d'une même session utilisent bien le même identifiant de sécurité.

A.1.33 Gestion des champs de sécurité du répertoire de clés du noyau

Compilation conditionnelle contrôlée par `CONFIG_KEYS`.

`key_alloc` Appelé pour contrôler la création d'une nouvelle clé dans le répertoire de clés et lui allouer une structure de sécurité.

`key_free` Appelé pour désallouer la structure de sécurité associée à une clé.

A.1.34 Contrôle des opérations effectuées sur le répertoire de clés du noyau

Compilation conditionnelle contrôlée par `CONFIG_KEYS`.

`key_permission` Appelé pour contrôler une opération sur une clé du registre de clés.

`key_getsecurity` Appelé pour sérialiser dans une chaîne de caractères le contexte de sécurité associée à une clé.

A.1.35 Gestion des champs de sécurité du système d'audit du noyau

Compilation conditionnelle contrôlée par `CONFIG_AUDIT`.

`audit_rule_init` Appelé pour initialiser la structure de sécurité associée à une nouvelle règle d'audit.

`audit_rule_free` Appelé lorsqu'une règle d'audit est déchargée, afin de désallouer la structure de sécurité correspondante.

A.1.36 Contrôle des opérations du système d'audit du noyau

Compilation conditionnelle contrôlée par `CONFIG_AUDIT`.

`audit_rule_known` Appelé pour savoir si une règle d'audit donnée contient des champs relatifs au module de sécurité.

`audit_rule_match` Appelé pour savoir si un contexte de sécurité donné correspond à une règle satisfaisant la vérification faite par `audit_rule_known`.

A.2 Crochets ajoutés pour Rfblare

Rfblare nécessite quelques crochets LSM supplémentaires comme détaillé dans les chapitres 5 et 6. Ils sont listés dans la table ci-après.

A.2.1 Contrôle de la fin d'un appel système provoquant un flux d'information

`syscall_before_return` Ce crochet est appelé lors du retour d'un appel système provoquant un flux discret comme `read`, `write`, ainsi que lorsqu'un appel système est interrompu par un signal.

A.2.2 Gestion des champs de sécurité des espaces d'adressage

`mm_dup_security` Ce crochet est appelé lors d'un appel à `clone` ou `fork` pour dupliquer la structure de sécurité attachée à l'espace d'adressage du processus appelant.

`mm_sec_free` Ce crochet décrémente le compteur de référence de la structure de sécurité de l'espace d'adressage du processus appelant et la désalloue si nécessaire. Il est appelé lorsqu'un thread disparaît.

A.2.3 Contrôle des opérations sur les files de message POSIX

`mq_store_msg` Ce crochet est appelé pour contrôler la réception d'un message depuis une file de messages POSIX.

Aucun crochet supplémentaire n'est nécessaire pour l'émission car le noyau utilise la même interface, donc les mêmes crochets, pour l'allocation des messages des files System V et POSIX.

A.2.4 Contrôle supplémentaire pour `ptrace`

`ptrace_unlink` Ce crochet est appelé lorsqu'un thread arrête de ptracer un autre.

Annexe B

Définition de la sémantique abstraite pour notre analyse statique et preuve de correction

Dans cette version imprimable du présent manuscrit, le script Coq complet de la preuve de correction de la sémantique abstraite nécessaire à l'analyse statique décrite dans le chapitre 5 n'est pas inclus. Il est disponible à l'adresse suivante : https://kayrebt.gforge.inria.fr/proofs.html.

Annexe C

Description formelle de la propagation de teintes de Rfblare et preuve de correction

Dans cette version imprimable du présent manuscrit, le script Coq complet donnant la description formelle de l'algorithme de propagation de teintes de Rfblare ainsi que sa preuve de correction n'est pas inclus. Il est disponible à l'adresse suivante : https://blare-ids.org/rfblare/.

Bibliographie

[1] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande et V. Viet Triem Tong. « GroddDroid : a gorilla for triggering malicious behaviors ». In : International Conference on Malicious and Unwanted Software (MALWARE 2015). Fajardo, Puerto Rico : IEEE, oct. 2015, p. 119–127. isbn : 978-1-5090-0317-4. doi : 10.1109/MALWARE.2015.7413692 (cf. p. 22).

[2] Radoniaina Andriatsimandefitra, Valérie Viet Triem Tong et Thomas Saliou. *Information Flow Policies vs Malware*. report. 16 sept. 2013 (cf. p. 22, 77).

[3] Arnd Bergmann. *[PATCH 20/20] BKL : That's all, folks [LWN.net]*. E-mail publié sur la liste de développement du noyau Linux. 25 jan. 2011. url : https://lwn.net/Articles/424677/ (visité le 07/05/2017).

[4] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala et Rupak Majumdar. « The Blast Query Language for Software Verification ». In : International Static Analysis Symposium (SAS 2004). Sous la dir. de Roberto Giacobazzi. T. 3148. Lecture Notes in Computer Science. Verona, Italy : Springer, août 2004, p. 2–18. isbn : 978-3-540-27864-1. doi : 10.1007/978-3-540-27864-1_2 (cf. p. 30, 126).

[5] Dirk Beyer et Alexander K. Petrenko. « Linux Driver Verification ». In : International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2012). Sous la dir. de Tiziana Margaria et Bernhard Steffen. T. 7610. Lecture Notes in Computer Science. Heraklion, Greece : Springer, oct. 2012, p. 1–6. isbn : 978-3-642-34031-4. doi : 10.1007/978-3-642-34032-1 (cf. p. 32).

[6] Daniel P. Bovet et Marco Cesati. *Understanding the Linux Kernel*. 3e éd. O'Reilly Media, nov. 2005. 944 p. isbn : 978-0-596-00565-8 (cf. p. 35, 55).

[7] Neil Brown. *Sparse : a look under the hood*. Linux Weekly News. 8 juin 2016. url : https://lwn.net/Articles/689907/ (visité le 01/02/2017) (cf. p. 29).

[8] T. Y. Chen, H. Leung et I. K. Mak. « Adaptive Random Testing ». In : *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*. Asian Computing Science Conference (ASIAN 2004). Sous la dir. de Michael J. Maher. T. 3321. Lecture Notes in Computer Science. DOI : 10.1007/978-3-540-30502-6_23. Chiang Mai, Thailand : Springer, 2004, p. 320–329. isbn : 978-3-540-24087-7. doi : 10.1007/978-3-540-30502-6_23 (cf. p. 31).

[9] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide et John Regehr. « Taming Compiler Fuzzers ». In : ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013). Seattle, WA, USA : ACM, 2013, p. 197–208. isbn : 978-1-4503-2014-6. doi : 10.1145/2491956.2462173 (cf. p. 31).

[10] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira et Barbara Liskov. « Abstractions for Usable Information Flow Control in Aeolus ». In : USENIX Annual Technical Conference (USENIX ATC 2012). Boston, MA, USA : USENIX Association, 2012, p. 139–151 (cf. p. 11–13, 17).

[11] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem et Dawson Engler. « An Empirical Study of Operating Systems Errors ». In : ACM Symposium on Operating Systems Principles (SOSP 2001). Banff, Alberta, Canada : ACM, 2001, p. 73–88. isbn : 978-1-58113-389-9. doi : 10.1145/502034.502042 (cf. p. 35).

[12] Kees Cook. *gcc-plugins : Add initial x86_64 kernexec plugin*. E-mail publié sur la liste de développement du noyau Linux. 13 jan. 2017. url : https://lwn.net/Articles/711655/ (visité le 06/02/2017).

[13] Kees Cook. *gcc-plugins : Add structleak for more stack initialization*. E-mail publié sur la liste de développement du noyau Linux. 13 jan. 2017. url : https://lwn.net/Articles/711692/ (visité le 06/02/2017).

[14] Jonathan Corbet. *Kernel building with GCC plugins*. Linux Weekly News. 14 juin 2014. url : https://lwn.net/Articles/691102/ (visité le 06/02/2017) (cf. p. 30).

[15] Jonathan Corbet, Jake Edge et Rebecca Sobol. *LWN.net News from the source*. Linux Weekly News. 1998. url : https://lwn.net (cf. p. 35).

[16] Maximiliano Cristiá et Pablo Enrique Mata. « Runtime enforcement of noninterference by duplicating processes and their memories ». In : Workshop de Seguridad Informática (WSegI 2009). T. 2009. 00008. Mar del Plata, Argentina, août 2009 (cf. p. 11, 77).

[17] Maximiliano Cristía et Pablo Enrique Mata. *Flowx : Implementación de no interferencia en Linux*. Rosario, Argentina : Facultad de Ciencias Exactas, Ingeniería y Agrimensura, Universidad Nacional de Rosario, 27 nov. 2009, p. 186 (cf. p. 13, 77).

[18] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman et F. Kenneth Zadeck. « Efficiently Computing Static Single Assignment Form and the Control Dependence Graph ». In : *ACM Transactions on Programming Languages and Systems* 13.4 (oct. 1991), p. 451–490. issn : 0164-0925. doi : 10.1145/115372.115320 (cf. p. 62, 65).

[19] Al Danial. *CLOC – Count Lines of Code*. 00021. 30 juil. 2014 (cf. p. 35).

[20] Dorothy E. Denning. « A Lattice Model of Secure Information Flow ». In : *Communications of the ACM* 19.5 (mai 1976), p. 236–243. issn : 0001-0782. doi : 10.1145/360051.360056 (cf. p. 7, 13).

[21] Dorothy Elizabeth Robling Denning et Peter J. Denning. « Certification of programs for secure information flow ». In : *Communications of the ACM* 20.7 (juil. 1977). Sous la dir. de Robert L. Ashenhurst, p. 504–513. issn : 0001-0782 (cf. p. 13).

[22] David Drysdale. *Coverage-guided kernel fuzzing with syzkaller*. Linux Weekly News. 2 mar. 2016. URL : https://lwn.net/Articles/677764/ (visité le 06/02/2017) (cf. p. 31).

[23] Bruno Dutertre et Leonardo de Moura. *The Yices SMT solver*. SRI International, 2006 (cf. p. 123).

[24] Jake Edge. *A pair of GCC plugins*. Linux Weekly News. 27 jan. 2017. URL : https://lwn.net/Articles/712161/ (visité le 06/02/2017) (cf. p. 30).

[25] Jake Edge. *The kernel address sanitizer*. Linux Weekly News. 14 sept. 2014. URL : https://lwn.net/Articles/612153/ (visité le 02/02/2017) (cf. p. 32).

[26] Antony Edwards et Trent Jaeger. « Maintaining the correctness of the Linux security modules framework ». In : *Ottawa Linux Symposium*. 2002, p. 223 (cf. p. 33).

[27] J. Ellson, E. R. Gansner, E. Koutsofios, S.C. North et G. Woodhull. « Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools ». In : *Graph Drawing Software*. Sous la dir. de M. Junger et P. Mutzel. Berlin/Heidelberg, Germany : Springer, 2004, p. 127–148. ISBN : 3-540-00881-0 (cf. p. 66).

[28] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel et Anmol N. Sheth. « TaintDroid : An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones ». In : USENIX Conference on Operating Systems Design and Implementation (OSDI 2010). Vancouver, BC, Canada : USENIX Association, 2010, p. 1–6 (cf. p. 12, 13, 77).

[29] J. S. Fenton. « Memoryless Subsystems ». In : *The Computer Journal* 17.2 (1er jan. 1974), p. 143–147. ISSN : 0010-4620. DOI : 10.1093/comjnl/17.2.143 (cf. p. 10).

[30] Jeffrey Scott Foster. « Type qualifiers : lightweight specifications to improve software quality ». Thèse de doct. Berkeley, CA, USA : University of California at Berkeley, 2002 (cf. p. 29).

[31] Tal Garfinkel. « Traps and Pitfalls : Practical Problems in System Call Interposition Based Security Tools. » In : Network and Distributed Systems Security Symposium (NDSS 2003). San Diego, CA, USA : The Internet Society, fév. 2003, p. 163–176. ISBN : 1-891562-16-9 (cf. p. 24, 26, 27).

[32] Samir Genaim et Fausto Spoto. « Information Flow Analysis for Java Bytecode ». In : International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005). Sous la dir. de Radhia Cousot. T. 3385. Lecture Notes in Computer Science. Paris, France : Springer, jan. 2005, p. 346–362. ISBN : 978-3-540-24297-0. DOI : 10.1007/978-3-540-30579-8_23 (cf. p. 11).

[33] Laurent George, Valérie Viet Triem Tong et Ludovic Mé. « Blare Tools : A Policy-Based Intrusion Detection System Automatically Set by the Security Policy ». In : International Workshop on Recent Advances in Intrusion Detection (RAID 2009). Sous la dir. d'Engin Kirda, Somesh Jha et Davide Balzarotti. T. 5758. Lecture Notes in Computer Science. Saint-Malo, France : Springer, sept. 2009, p. 355–356. ISBN : 978-3-642-04341-3. DOI : 10.1007/978-3-642-04342-0_22 (cf. p. 11, 13, 21, 73, 77, 129).

[34] Laurent Georget, Mathieu Jaume, Guillaume Piolle, Frédéric Tronel et Valérie Viet Triem Tong. « Information Flow Tracking for Linux Handling Concurrent System Calls and Shared Memory ». In : *Software Engineering and Formal Methods*. Software Engineering & Formal Methods (SEFM 2017). Sous la dir. d'Alessandro Cimatti et Marjan Sirjani. T. 10469. Lecture Notes in Computer Science. Trento, Italy : Springer, Cham, Switzerland, 4 sept. 2017, p. 1–16. isbn : 978-3-319-66196-4. doi : 10.1007/978-3-319-66197-1_1 (cf. p. 148).

[35] Laurent Georget, Mathieu Jaume, Guillaume Piolle, Frédéric Tronel et Valérie Viet Triem Tong. « Suivi de flux d'information correct sous Linux ». In : Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2017). Sous la dir. d'Akram Idani et Nikolai Kosmatov. Montpellier, France, juin 2017.

[36] Laurent Georget, Mathieu Jaume, Guillaume Piolle, Frédéric Tronel et Valérie Viet Triem Tong. « Verifying the Reliability of Operating System-Level Information Flow Control Systems in Linux ». In : FME Workshop on Formal Methods in Software Engineering (FormaliSE 2017). Buenos Aires, Argentina : IEEE Press, mai 2017, p. 10–16. isbn : 978-1-5386-0422-9. doi : 10.1109/FormaliSE.2017..1 (cf. p. 126).

[37] Laurent Georget, Frédéric Tronel et Valérie Viet Triem Tong. « Kayrebt : An Activity Diagram Extraction and Visualization Toolset Designed for the Linux Codebase ». In : IEEE Working Conference on Software Visualization (VISSOFT 2015). Bremen, Germany : IEEE, sept. 2015, p. 170–174. doi : 10.1109/VISSOFT.2015.7332431 (cf. p. 71).

[38] Christophe Hauser. « Détection d'intrusion dans les systèmes distribués par propagation de teinte au niveau noyau ». Doctoral thesis. Rennes, France : University of Rennes 1, juin 2013 (cf. p. 11, 20, 21, 75, 82, 132, 133).

[39] Christophe Hauser et Guillaume Brogi. *KBlare*. 2014. Logiciel. url : https://git.blare-ids.org (visité le 10/04/2017).

[40] Guillaume Hiet. « Détection d'intrusions paramétrée par la politique de sécurité grâce au contrôle collaboratif des flux d'informations au sein du système d'exploitation et des applications : mise en œuvre sous Linux pour les programmes Java ». Thèse de doct. Rennes, France : Supélec, 2008 (cf. p. 11, 20, 21, 77).

[41] D. Richard Hipp, Dan Kennedy et Joe Mistachkin. *SQLite*. Version 3.18.0, publiée le 30 mar. 2017. Logiciel. url : https://www.sqlite.org (visité le 07/05/2017).

[42] Trent Jaeger, Antony Edwards et Xiaolan Zhang. « Consistency analysis of authorization hook placement in the Linux security modules framework ». In : *ACM Trans. Inf. Syst. Secur.* 7.2 (2004), p. 175–205 (cf. p. 32, 150).

[43] Sushil Jajodia, Ravi S. Sandhu et Barbara T. Blaustein. « Solutions to the Polyinstanciation Problem ». In : *Information Security : An integrated Collection of Essays*. Sous la dir. de Marshall D. Abrams, Sushil Jajodia et Harold J. Podell. Los Alamitos, CA, USA : IEEE Computer Society Press, 1995, p. 493–530. isbn : 0-8186-3662-9 (cf. p. 15).

[44] Mathieu Jaume. « Semantic Comparison of Security Policies : From Access Control Policies to Flow Properties ». In : IEEE Symposium on Security and Privacy Workshops (SPW 2012). San Francisco, CA, USA : IEEE, mai 2012, p. 60–67. ISBN : 978-1-4673-2157-0. DOI : 10.1109/SPW.2012.33 (cf. p. 7).

[45] Mathieu Jaume, Radoniaina Andriatsimandefitra, Valérie Viet Triem Tong et Ludovic Mé. « Secure states versus Secure executions : From access control to flow control ». In : International Conference on Information Systems Security (ICISS 2013). T. 8303. Lecture Notes in Computer Science. Calcutta, India : Springer, déc. 2013, p. 148–162. ISBN : 978-3-642-45203-1. DOI : 10.1007/978-3-642-45204-8_11 (cf. p. 7).

[46] Mathieu Jaume, Valérie Viet Triem Tong et Ludovic Mé. « Flow based interpretation of access control : Detection of illegal information flows ». In : International Conference on Information Systems Security (ICISS 2011). T. 7093. Lecture Notes in Computer Science. Kolkata, India : Springer, déc. 2011, p. 72–86. ISBN : 978-3-642-25559-5. DOI : 10.1007/978-3-642-25560-1_5 (cf. p. 7, 8).

[47] Rob Johnson et David Wagner. « Finding User/Kernel Pointer Bugs with Type Inference ». In : USENIX Security Symposium (USENIX 2004). T. 13. San Diego, CA, USA : USENIX Association, 2004 (cf. p. 29).

[48] Dave Jones. *Trinity*. Version v1.7, publiée le 28 oct. 2016. Logiciel. URL : https://github.com/kernelslacker/trinity (visité le 06/02/2017).

[49] Michael Kerrisk. *LCA : The Trinity fuzz tester*. Linux Weekly News. 6 fév. 2013. URL : https://lwn.net/Articles/536173/ (visité le 06/02/2017) (cf. p. 31).

[50] Alexey Khoroshilov, Vadim Mutilin, Alexander Petrenko et Vladimir Zakharov. « Establishing Linux Driver Verification Process ». In : *Perspectives of Systems Informatics*. International Andrei Ershov Memorial Conference on Perspectives of System Informatics. T. 5947. Lecture Notes in Computer Science. DOI : 10.1007/978-3-642-11486-1_14. Novosibirsk, Russia : Springer, juin 2009, p. 165–176. ISBN : 978-3-642-11485-4. DOI : 10.1007/978-3-642-11486-1_14 (cf. p. 32).

[51] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, Marinus Frans Kaashoek, Eddie Kohler et Robert Morris. « Information flow control for standard OS abstractions ». In : ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2007). Stevenson, WA, USA : ACM, oct. 2007, p. 321–334. ISBN : 978-1-59593-591-5. DOI : 10.1145/1294261.1294293 (cf. p. 11, 13, 77).

[52] Leonard J. LaPadula et D. Elliott Bell. *Secure Computer Systems : A Mathematical Model*. MTR-2547 (ESD-TR-73-278-II) Vol. 2. Bedford : MITRE Corp., mai 1973 (cf. p. 6).

[53] Donald C. Latham. *Trusted Computer System Evaluation Criteria (Orange Book)*. 5200.28-STD. Arlington County, VA, USA : Department of Defense, déc. 1985, p. 116 (cf. p. 14).

[54] Robert Love. *Linux Kernel Development*. 3e éd. Upper Saddle River, NJ : Addison Wesley, 2 juil. 2010. 440 p. ISBN : 978-0-672-32946-3 (cf. p. 35, 55).

[55] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman et W. R. Shockley. « The SeaView security model ». In : *IEEE Transactions on Software Engineering* 16.6 (juin 1990), p. 593–607. ISSN : 0098-5589. DOI : 10.1109/32.55088 (cf. p. 15).

[56] M. Douglas McIlroy et James A. Reeds. « Multilevel Security in the UNIX Tradition ». In : *Software : Practice and Experience* 22.8 (1992), p. 673–694. ISSN : 0038-0644. DOI : 10.1002/spe.4380220805 (cf. p. 10, 13, 14, 140).

[57] M. Mendonça et N. Neves. « Fuzzing Wi-Fi Drivers to Locate Security Vulnerabilities ». In : European Dependable Computing Conference (EDCC 2008). Khaunas, Lithuania : IEEE, mai 2008, p. 110–119. ISBN : 978-0-7695-3138-0. DOI : 10.1109/EDCC-7.2008.22 (cf. p. 31).

[58] Jason Merrill. « GENERIC and GIMPLE : A new tree representation for entire functions ». In : *GCC Developers Summit*. 2003, p. 171–180 (cf. p. 62).

[59] Barton P. Miller, Louis Fredriksen et Bryan So. « An Empirical Study of the Reliability of UNIX Utilities ». In : *Communications of the ACM* 33.12 (déc. 1990), p. 32–44. ISSN : 0001-0782. DOI : 10.1145/96267.96279 (cf. p. 31).

[60] Justin J. Miller. « Graph database applications and concepts with Neo4j ». In : *Southern Association for Information Systems Conference Proceedings*. Southern Association for Information Systems Conference. T. 24. Atlanta, GA, USA : Association for Information Systems, mar. 2013 (cf. p. 71).

[61] Jan Tobias Mühlberg et Gerald Lüttgen. « BLASTing linux code ». In : International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2006). Sous la dir. de Luboš Brim, Boudewijn Haverkort, Martin Leucker et Jaco van de Pol. T. 4346. Lecture Notes in Computer Science. Bonn, Germany : Springer, 2006, p. 211–226. ISBN : 978-3-540-70951-0. DOI : 10.1007/978-3-540-70952-7_14 (cf. p. 30).

[62] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao et Gerwin Klein. « seL4 : From General Purpose to a Proof of Information Flow Enforcement ». In : IEEE Symposium on Security and Privacy (S&P 2013). Berkeley, CA, USA : IEEE, mai 2013, p. 415–429. ISBN : 978-0-7695-4977-4. DOI : 10.1109/SP.2013.35 (cf. p. 11).

[63] Divya Muthukumaran, Trent Jaeger et Vinod Ganapathy. « Leveraging "choice" to automate authorization hook placement ». In : 00006. ACM Press, 2012, p. 145. ISBN : 978-1-4503-1651-4. DOI : 10.1145/2382196.2382215 (cf. p. 33).

[64] Andrew C. Myers. « JFlow : Practical Mostly-static Information Flow Control ». In : *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Antonio, Texas, USA : ACM, 1999, p. 228–241. ISBN : 1-58113-095-3. DOI : 10.1145/292540.292561 (cf. p. 11, 13, 17).

[65] Andrew C. Myers et Barbara Liskov. « A Decentralized Model for Information Flow Control ». In : ACM Symposium on Operating Systems Principles (SOSP 1997). Saint-Malo, France : ACM, 1997, p. 129–142. ISBN : 978-0-89791-916-6. DOI : 10.1145/268998.266669 (cf. p. 12, 15, 17).

[66] Andrew C. Myers et Barbara Liskov. « Protecting Privacy Using the Decentralized Label Model ». In : *ACM Transactions on Software Engineering Methodology* 9.4 (oct. 2000), p. 410–442. ISSN : 1049-331X. DOI : 10.1145/363516.363526 (cf. p. 11–13, 17).

[67] Adwait Nadkarni, Benjamin Andow, William Enck et Somesh Jha. « Practical DIFC Enforcement on Android ». In : USENIX Security Symposium (USENIX 2016). Austin, TX, USA : USENIX Association, août 2016, p. 1119–1136. isbn : 978-1-931971-32-4 (cf. p. 11, 13, 21, 73, 77, 129).

[68] Amon Ott. *RSBAC and LSM*. RSBAC : Extending Linux Security Beyond the Limits. 2006. url : https://www.rsbac.org/documentation/why_rsbac_does_not_use_lsm (visité le 10/02/2017) (cf. p. 73).

[69] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen et Gilles Muller. « Documenting and Automating Collateral Evolutions in Linux Device Drivers ». In : European Conference on Computer Systems (EuroSys 2008). Glasgow, Scotland : ACM, avr. 2008, p. 247–260 (cf. p. 29).

[70] Perl5 Porters, éd. *perlsec - Perl Manual*. Version Perl 5 version 24. Mai 2016 (cf. p. 10).

[71] Hendrik Post, Carsten Sinz et Wolfgang Küchlin. « Towards automatic software model checking of thousands of Linux modules—a case study with Avinux ». In : *Software Testing, Verification and Reliability* 19.2 (2009), p. 155–172 (cf. p. 32).

[72] François Pottier et Vincent Simonet. « Information Flow Inference for ML ». In : *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25.1 (2003), p. 117–158. issn : 0164-0925. doi : 10.1145/596980.596983 (cf. p. 11).

[73] Q-Success. *Usage Statistics and Market Share of Linux for Websites, March 2017*. W3Techs Web Technology Surveys. 2016. url : https://w3techs.com/technologies/details/os-linux/all/all (visité le 14/03/2017) (cf. p. 36).

[74] Henry Gordon Rice. « Classes of Recursively Enumerable Sets and Their Decision Problems ». In : *Transactions of the American Mathematical Society* 74.2 (1953), p. 358–366. issn : 00029947. doi : 10.2307/1990888 (cf. p. 28, 94).

[75] Indrajit Roy et Don Porter. *Laminar*. Version v2, publiée le 20 août 2014. Logiciel. url : https://sourceforge.net/p/jikesrvm/research-archive/26 (visité le 22/02/2017).

[76] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley et Emmett Witchel. « Laminar : Practical Fine-grained Decentralized Information Flow Control ». In : ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009). Dublin, Ireland : ACM, juin 2009, p. 63–74. isbn : 978-1-60558-392-1. doi : 10.1145/1543135.1542484 (cf. p. 11, 13, 19, 20, 73, 77, 129).

[77] Paul Rusty Russell. « Unreliable guide to hacking the Linux kernel ». 2000 (cf. p. 30, 57).

[78] Andrey Ryabinin. *UBSan : run-time undefined behavior sanity checker*. E-mail publié sur la liste de développement du noyau Linux. 20 oct. 2014. url : https://lwn.net/Articles/617364/ (visité le 02/02/2017).

[79] Jerome. H. Saltzer et Michael. D. Schroeder. « The Protection of Information in Computer Systems ». In : *Proceedings of the IEEE* 63.9 (sept. 1975), p. 1278–1308. issn : 0018-9219. doi : 10.1109/PROC.1975.9939 (cf. p. 10).

[80] Randal L. Schwartz, Brian D. Foy et Tom Phoenix. *Learning Perl*. 6e éd. O'Reilly Media, juin 2011. 388 p. isbn : 978-1-4493-0358-7 (cf. p. 10).

[81] P. E. Shved, V. S. Mutilin et M. U. Mandrykin. « Experience of improving the blast static verification tool ». In : *Programming and Computer Software* 38.3 (1er juin 2012), p. 134–142. issn : 0361-7688, 1608-3261. doi : 10.1134/S0361768812030061 (cf. p. 30).

[82] K. Y. Sim, F.-C. Kuo et R. Merkel. « Fuzzing the Out-of-memory Killer on Embedded Linux : An Adaptive Random Approach ». In : ACM Symposium on Applied Computing (SAC 2011). TaiChung, Taiwan : ACM, 2011, p. 387–392. isbn : 978-1-4503-0113-8. doi : 10.1145/1982185.1982268 (cf. p. 31).

[83] Jiří Slabý. « Automatic Bug-finding Techniques for Large Software Projects ». Doctoral thesis. Brno, Czech Republic : Masaryk University, Faculty of Informatics, 4 mar. 2014 (cf. p. 29).

[84] Stephen Smalley. *[PATCH 0/2] Add missing LSM hooks in mq_timed{send,receive} and splice, Email on the Linux Security Modules development mailing list.* E-mail publié sur la liste de développement du noyau Linux. Juil. 2016. url : http://thread.gmane.org/gmane.linux.kernel.lsm/28737.

[85] Stephen Smalley, Chris Vance et Wayne Salamon. *Implementing SELinux as a Linux security module.* 01-043. Santa Clara, CA, USA : Network Associates Inc., 2001, p. 139 (cf. p. 73).

[86] Brad Spender. « SSTIC 2016 Keynote - grsecurity ». Symposium sur la sécurité des technologies de l'information et des communications. Rennes, France, 1er juin 2006 (cf. p. 30).

[87] Brad Spender. *Why doesn't grsecurity use LSM ?* grsecurity. 2008. url : https://grsecurity.net/lsm.php (visité le 10/02/2017) (cf. p. 73).

[88] Richard Matthew Stallman et the GCC developer community. *Plugins - GNU Compiler Collection (GCC) Internals.* 2015. url : https://gcc.gnu.org/onlinedocs/gccint/Plugins.html#Plugins (visité le 18/05/2015) (cf. p. 61).

[89] Richard Matthew Stallman et the GCC developer community. *Using the GNU Compiler Collection (GCC).* 2013 (cf. p. 30, 61, 190).

[90] StatCounter. *StatCounter Global Stats - Browser, OS, Search Engine including Mobile Usage Share.* StatCounter Global Stats. Nov. 2016. url : http://gs.statcounter.com/ (visité le 14/03/2017) (cf. p. 36).

[91] Erich Strohmaier, Jack Dongarra, Horst Simon et Martin Meuer. *Operating system Family / Linux.* TOP500 Supercomputer Sites. Nov. 2016. url : https://www.top500.org/statistics/details/osfam/1 (visité le 14/03/2017) (cf. p. 36).

[92] Technical Committee X3J11. *American National Standard for Information Systems — Programming Language C.* X3.159-1989. American National Standard Institute, 1989 (cf. p. 57).

[93] The Coq Development Team. *The Coq Proof Assistant Reference Manual.* 14 déc. 2016 (cf. p. 88, 129).

[94] Craig Timberg. « The Kernel of the Argument ». In : *Washington Post* (5 nov. 2015) (cf. p. 36).

[95] Linus Torvalds. *Re : [Regression w/ patch] Media commit causes user space to misbahave (was : Re : Linux 3.8-rc1)*. E-mail publié sur la liste de développement du noyau Linux. 23 déc. 2012. URL : https://lkml.org/lkml/2012/12/23/75 (visité le 28/02/2017).

[96] Linus Torvalds. *Sparse "context" checking.*. E-mail publié sur la liste de développement du noyau Linux. 30 oct. 2004. URL : https://lwn.net/Articles/109066/ (visité le 03/02/2017).

[97] Linus Torvalds. *The Linux Kernel*. Version 4.7, publiée le 24 juil. 2016. Logiciel. Linux Kernel Organization, Inc. URL : https://kernel.org (visité le 02/02/2017).

[98] Linus Torvalds, Josh Triplett et Christopher Li. *Sparse – a semantic parser for C*. 2003. Logiciel. URL : https://sparse.wiki.kernel.org.

[99] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani et David I. August. « RIFLE : An Architectural Framework for User-Centric Information-Flow Security ». In : International Symposium on Microarchitecture (MICRO 2004). Portland, OR, USA : IEEE, 2004, p. 243–254. ISBN : 0-7695-2126-6. DOI : 10.1109/MICRO.2004.31 (cf. p. 11).

[100] Steve VanDeBogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris et David Mazières. « Labels and Event Processes in the Asbestos Operating System ». In : *ACM Transactions on Computer Systems* 25.4 (déc. 2007). ISSN : 07342071. DOI : 10.1145/1314299.1314302 (cf. p. 11, 13, 14).

[101] Valérie Viet Triem Tong, Andrew Clark et Ludovic Mé. « Specifying and Enforcing a Fined-Grained Information Flow Policy : Model and Experiments ». In : *Mist*. 2010 (cf. p. 20).

[102] Clark Weissman. « Security Controls in the ADEPT-50 Time-sharing System ». In : Fall Joint Computer Conference (AFIPS 1969). Las Vegas, NV, USA : ACM, nov. 1969, p. 119–133. DOI : 10.1145/1478559.1478574 (cf. p. 5–7).

[103] Thomas Witkowski, Nicolas Blanc, Daniel Kroening et Georg Weissenbacher. « Model Checking Concurrent Linux Device Drivers ». In : IEEE/ACM International Conference on Automated Software Engineering (ASE 2007). Atlanta, GA, USA : ACM, 2007, p. 501–504. ISBN : 978-1-59593-882-4. DOI : 10.1145/1321631.1321719 (cf. p. 32).

[104] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley et Greg Kroah-Hartman. « Linux Security Module Framework ». In : Ottawa Linux Symposium. Ottawa, Ontario, Canada, 2002 (cf. p. 19, 73).

[105] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris et Greg Kroah-Hartman. « Linux Security Modules : General Security Support for the Linux Kernel ». In : USENIX Security Symposium (USENIX 2002). San Francisco, CA, USA : USENIX Association, 2002, p. 17–31. ISBN : 1-931971-00-5 (cf. p. 19, 27, 73).

[106] Lok Kwong Yan et Heng Yin. « DroidScope : Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis ». In : USENIX Security Symposium (USENIX 2012). Bellevue, WA, USA : USENIX Association, août 2012, p. 29–29 (cf. p. 12, 22).

[107] Junfeng Yang, Ted Kremenek, Yichen Xie et Dawson Engler. « MECA : an extensible, expressive system and language for statically checking security properties ». In : ACM conference on Computer and Communications Security (CCS 2003). Washington D.C., USA : ACM, oct. 2003, p. 321–334 (cf. p. 29).

[108] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel et Engin Kirda. « Panorama : Capturing System-wide Information Flow for Malware Detection and Analysis ». In : *ACM Conference on Computer and Communications Security*. Alexandria, Virginia, USA : ACM, 2007, p. 116–127. isbn : 978-1-59593-703-2. doi : 10.1145/1315245.1315261 (cf. p. 11, 12).

[109] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler et David Mazières. « Making Information Flow Explicit in HiStar ». In : Symposium on Operating Systems Design and Implementation (OSDI 2006). Seattle, WA, USA : USENIX Association, nov. 2006, p. 263–278. isbn : 1-931971-47-1 (cf. p. 11, 13, 16, 140).

[110] Nickolai Zeldovich, Silas Boyd-Wickizer et David Mazières. « Securing Distributed Systems with Information Flow Control. » In : *NSDI'08 : Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. T. 8. Berkeley, CA, USA : USENIX Association, 2008, p. 293–308 (cf. p. 11).

[111] Nickolai Zeldovich, Hari Kannan, Michael Dalton et Christos Kozyrakis. « Hardware Enforcement of Application Security Policies Using Tagged Memory ». In : USENIX Conference on Operating Systems Design and Implementation (OSDI 2008). San Diego, CA, USA : USENIX Association, déc. 2008, p. 225–240 (cf. p. 11).

[112] Xiaolan Zhang, Antony Edwards et Trent Jaeger. « Using CQUAL for Static Analysis of Authorization Hook Placement ». In : USENIX Security Symposium (USENIX 2002). San Francisco, CA, USA : USENIX Association, août 2002, p. 33–48. isbn : 1-931971-00-5 (cf. p. 32).

[113] Jacob Zimmermann. « Détection d'intrusions paramétrée par la politique par contrôle de flux de références ». Thèse de doct. Université de Rennes 1, 16 déc. 2003 (cf. p. 20).

[114] Jacob Zimmermann, Ludovic Mé et Christophe Bidan. « An Improved Reference Flow Control Model for Policy-Based Intrusion Detection ». In : *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS)*. Oct. 2003 (cf. p. 20).

[115] Jacob Zimmermann, Ludovic Mé et Christophe Bidan. « Experimenting with a Policy-Based HIDS Based on an Information Flow Control Model ». In : *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. Déc. 2003 (cf. p. 21).

Publications de l'auteur associées aux contributions de cette thèse

Conférences internationales avec actes

[34] Laurent Georget, Mathieu Jaume, Guillaume Piolle, Frédéric Tronel et Valérie Viet Triem Tong. « Information Flow Tracking for Linux Handling Concurrent System Calls and Shared Memory ». In : *Software Engineering and Formal Methods*. Software Engineering & Formal Methods (SEFM 2017). Sous la dir. d'Alessandro Cimatti et Marjan Sirjani. T. 10469. Lecture Notes in Computer Science. Trento, Italy : Springer, Cham, Switzerland, 4 sept. 2017, p. 1–16. isbn : 978-3-319-66196-4. doi : 10.1007/978-3-319-66197-1_1 (cf. p. 148).

[36] Laurent Georget, Mathieu Jaume, Guillaume Piolle, Frédéric Tronel et Valérie Viet Triem Tong. « Verifying the Reliability of Operating System-Level Information Flow Control Systems in Linux ». In : FME Workshop on Formal Methods in Software Engineering (FormaliSE 2017). Buenos Aires, Argentina : IEEE Press, mai 2017, p. 10–16. isbn : 978-1-5386-0422-9. doi : 10.1109/FormaliSE.2017..1 (cf. p. 126).

[37] Laurent Georget, Frédéric Tronel et Valérie Viet Triem Tong. « Kayrebt : An Activity Diagram Extraction and Visualization Toolset Designed for the Linux Codebase ». In : IEEE Working Conference on Software Visualization (VISSOFT 2015). Bremen, Germany : IEEE, sept. 2015, p. 170–174. doi : 10.1109/VISSOFT.2015.7332431 (cf. p. 71).

Conférence française avec actes

[35] Laurent Georget, Mathieu Jaume, Guillaume Piolle, Frédéric Tronel et Valérie Viet Triem Tong. « Suivi de flux d'information correct sous Linux ». In : Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2017). Sous la dir. d'Akram Idani et Nikolai Kosmatov. Montpellier, France, juin 2017.

Table des figures

Liste des tableaux

Liste des extraits de code

Glossaire

daemon Programme s'exécutant en tâche de fond et exécutant des tâches d'administration en continu. 20

firmware Code fourni par un fabricant de périphérique matériel, destiné à être chargé sur le périphérique à son initialisation. Il se distingue du pilote en ce qu'il est absolument nécessaire au fontionnement du périphérique et s'exécute sur celui-ci, et non dans le système d'exploitation. Il s'agit typiquement du micro-code de composants intégrés programmables. 37

initramfs Système de fichiers de chargement du noyau, mini-système d'exploitation exécuté par le système de démarrage de l'ordinateur pour monter le système de fichiers racine, et décompresser et démarrer le vrai noyau. 37

patch Correctif, fichier se présentant comme une liste de différence entre un ensemble de fichiers de code source et ces mêmes fichiers « corrigés », par exemple, pour ajouter une fonctionnalité ou éliminer un *bug*. Le développement du noyau est organisé autour de ces *patchs* : les développeurs voulant proposer leur contribution au noyau le font sous la forme de *patchs* postés sur une liste de diffusion où ils sont commentés, discutés, retravaillés et finalement acceptés ou refusés par le mainteneur en charge de la partie du noyau concernée. 37, 126

Process IDentifier Identifiant de ce qui est en réalité une tâche pour le noyau Linux, et qui correspond donc plutôt à un *thread* du point de vue de l'espace utilisateur. Le nom subsiste de l'époque où Linux ne gérait pas le *multithreading* dans le noyau. 190

Thread Group IDentifier Identifiant de groupe de *threads*, ce que l'on nommerait plus couramment « processus ». 190

thread Fil d'exécution, plus petite entité pouvant être ordonnancée par le noyau pour exécuter du code. Un processus est composé de plusieurs *threads* partageant un même gestionnaire de signaux (et pour des raisons de commodité le même espace d'adressage). 10, 16, 18–21, 37, 40–45, 47, 55, 58, 59, 68, 74, 75, 80, 126, 189

vanilla Le noyau Linux *vanilla* désigne le noyau « officiel », tel que délivré par Linux Torvalds, le mainteneur officiel, et son équipe, sans modification apportée par une tierce partie. Le nom vient de ce que certaines personnes disent qu'un aliment nature est « à la vanille » même quand il n'en contient pas, pour une raison mystérieuse. 21, 75, 152, 153

API Application Programming Interface. 13, 37, 49, 67, *Voir :* Application Programming Interface

Application Programming Interface Interface de programmation applicative, module d'une bibliothèque, d'un service web, d'un système d'exploitation, etc.

développé, documenté et maintenu dans le but de permettre à un programme d'en utiliser les fonctionnalités. 189

Decentralized Information Flow Control Contrôle de flux d'information décentralisé, où chaque utilisateur peut décider au moins en partie de la politique à appliquer sur les données dont il est propriétaire dans le système. 12, 190

Department of Defense Équivalent états-unien du ministère de la défense. 5, 190

DIFC Decentralized Information Flow Control. 12, 15, 17, 22, *Voir :* Decentralized Information Flow Control

DoD Department of Defense. 5, 6, 14, *Voir :* Department of Defense

GCC *Gnu Compilers Collection* [89]. 30, 32, 57, 58, 60–63, 65, 66, 70, 72, 82, 83, 87–89, 92, 95, 122, 123, 126, 151

Inter-Process Communication Canal de communication que deux processus ou plus peuvent partager pour s'achanger des données et collaborer, comme par exemple les tuyaux (*pipes*), les sockets réseaux, etc. 13, 49, 190

IP *Internet Protocol*. 11, 53–55

IPC Inter-Process Communication. 13, 17, 21, 22, 37, 44, 49, 51, 53, 78, 160, *Voir :* Inter-Process Communication

Linux Security Modules Modules de sécurité Linux. Désigne à la fois un *framework* conçu pour faciliter le développement et l'intégration de modules de sécurité dans le noyau Linux ainsi que ces modules eux-mêmes. 2, 190

LSM Linux Security Modules. 2, 3, 19–21, 27, 28, 32, 33, 37–41, 45, 46, 54, 57, 67, 73–75, 77, 78, 80–87, 104, 117, 122–128, 130, 136, 139, 144–146, 148, 150–153, 155, 165, *Voir :* Linux Security Modules

PID *Process IDentifier*. 44, *Voir : Process IDentifier*

Portable Operating System Interface Norme publiée à l'origine par IEEE, puis pas l'Open Group, standardisant les interfaces et utilitaires de base qu'un système d'exploitation devrait fournir pour garantir leur compatibilité et leur « UNIXité ». 49, 190

POSIX Portable Operating System Interface. 49, 51–53, *Voir :* Portable Operating System Interface

Remote Procedure Call Appel de fonction à distance. Mécanisme permettant à un processus d'appeler une fonction d'un programme s'exécutant dans un processus distinct. 18, 190

RPC Remote Procedure Call. 18, *Voir :* Remote Procedure Call

TGID *Thread Group IDentifier*. *Voir : Thread Group IDentifier*

treillis Un treillis est un ensemble équipé d'une relation d'ordre tel que toute paire d'éléments admet un unique plus petit élément supérieur et un unique plus grand élément inférieur.

Si l'ensemble est fini, le treillis également, et il existe alors un unique élément minimal et un unique élément maximal.

On note couramment $\langle E, \preceq, \sqcap, \sqcup, \top, \bot \rangle$ avec :

- E : l'ensemble considéré ;
- $\preceq$: l'ordre (généralement partiel, sinon le treillis est dégénéré) ;
- $\sqcap$: l'opération *join*, donnant le plus petit élément supérieur de deux éléments ;
- $\sqcup$: l'opération *meet*, donnant le plus grand élément inférieur à deux éléments ;
- $\top$: l'élément maximal de E (si E est fini) ;
- $\bot$: l'élément minimal de E (si E est fini).

7

UTS *UNIX Time-Sharing*. 44